MECHANISMS FOR PROVABLE INTEGRITY PROTECTION IN DECENTRALIZED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ethan Benjamin Cecchetti

August 2021

MECHANISMS FOR PROVABLE INTEGRITY PROTECTION IN DECENTRALIZED SYSTEMS

Ethan Benjamin Cecchetti, Ph.D.

Cornell University 2021

Decentralized systems are built from a set of coordinating independent services. Yet these services might not trust each other, making it difficult to maintain the integrity of the whole application. This dissertation explores two different approaches to achieving provable integrity guarantees in such systems.

The first technique, realized in Solidus, applies cryptographic tools to provably preserve the integrity of a blockchain-based financial transaction system while hiding the sender, receiver, and value of each transaction. The second complements the cryptographic approach by showing how to achieve strong integrity guarantees for realistic systems using language-based Information Flow Control (IFC). Traditional IFC systems only provide strong integrity guarantees in the absence of endorsement—treating inputs as more trusted than their source—but endorsement is necessary in real-world systems. This work classifies two ways in which unrestricted endorsements can compromise system integrity if attackers violate implicit assumptions. In both cases, IFC ideas help define security and support language-based rules to provably eliminate all attacks in the class.

BIOGRAPHICAL SKETCH

Ethan Cecchetti was born in Columbus, Ohio in 1990. He moved to Massachusetts in 2003 and graduated from Lexington High School five years later. He then attended Brown University where he earned a joint Bachelors degree in Mathematics and Computer Science in May 2012. Upon graduating, Ethan took a job working as a software engineer for the travel website TripAdvisor. He decided that, after three years out of school, he missed solving mathematical puzzles and was interested in the challenge of research, so he joined Cornell to pursue his PhD in August 2015.

For my grandparents, Elizabeth, Giovanni, Elliot, and Gloria.

I wish you were here to celebrate this with me.

ACKNOWLEDGEMENTS

A PhD is given to one person, but nobody goes through graduate school one alone. In normal times, this dissertation would have been impossible without the help of a great many people. Completing the final year during a global pandemic required help from even more.

First and foremost, to my parents, Ruth and Steve: Thank you for your unwavering love and support. I am privileged to have parents who understand the triumphs and disappointments of graduate life, and I relied on you heavily throughout. To my brother Dan and my entire extended family, you have always been interested in what I do, even if you don't understand it. I couldn't ask for more.

To my advisors, Andrew Myers and Ari Juels: You both pushed me to look at research problems from numerous directions and helped me see more angles than I knew existed. You helped to understand the how of research, not just the what. I may not have always appreciated your insistance that I figure things out for myself, but I am a better researcher for it.

Several other people deserve special thanks. Andrew Hirsch, Eston Schweickart, and Isaac Sheff made me feel like I had a real home in Ithaca. Sorry I chased you all off the continent. Rachit Nigam (who beautifully suggested I title this dissertation "Information Security go Brrr"), Vedant Puri, and Armin Namavari provided companionship and understanding as our lives were upended by a global pandemic. Thanks to all six of you for putting up with my nonsense around the house. Mae Milano provided a familiar face when I arrived and introduced me to more wonderful people and places than I can count as I built a life in Ithaca. Xanda Schofield and Jake and Katie Gardner made me feel loved and included when I needed friends. I cannot thank you three enough and I feel honored to be your friend. Elizabeth Sibert has always been there from afar, sharing the ups and

downs. I immensely appreciate your support, understanding, and ability to make our conversations feel comfortable and familiar, whether I ranted at you yesterday or we haven't spoken in months. Finally, Tegan Wilson arrived as many of my friends were graduating and moving away. I'm still amazed at how fast you became my best friend, and you didn't stop there. I cannot express how much you have improved my life in the last three years.

I also want to thank the entire Cornell Computing and Information Sciences community. First, Becky Stewart, Tammy Gardner, and the entire rest of the administrative staff of the department made many things easy that would have otherwise been impossible. Second, I am grateful for all of the friends I made in the CS department. I chose to come to Cornell because I believed I would have friends here, and I was not disappointed. Danny Adams had a knack for making it easy to open up about the difficulties of grad school, a skill he demonstrated reliably at his Tea & Treats events. Lorenzo Alvisi routinely made me feel welcome and brightened my day, simply by being excited to see me. Shrutarshi Basu's seamless transitions between deadpan jokes, fountain pen recommendations, and deep conversations about life came exactly when I needed them. And Soumya Basu was, of course, a great backup when the other Basu wasn't around. Eleanor Birrell made sure I knew where to get the best food and drinks in town. The incessant silliness of both Eric Campbell and Jonathan DiLorenzo constantly reminded me to enjoy myself. Saying hi to Claire Cardie and her wonderful dogs, first Marseille and later Mirabelle and Mayenne, was exactly what I needed on many stressful afternoons. Natacha Crooks' terrible horticulture skills never stopped her from being a great friend and collaborator. Molly Q Feldman kept me sane by letting me interrupt her several times a week with my triumphs and failures, great and small. As my student mentor, Tom Magrino helped me navigate finding an advisor and start-

ing research and became a great friend along the way. Fabian Muehlboeck was a constant source of interesting conversations and delicious baked goods, though I think I've had enough gummy bears. Veronica VanCleave-Seeley's door was always open if I needed a break or pictures of her adorable cat Kaladin, whom I finally confirmed is just as cute in person. Too many other students, faculty, and staff to name provided the friendly faces and fun conversations that made me want to come to the office every day.

Thankfully, my life did not stop at the doors of Gates Hall. Whether introducing me to a varied cast of new friends, giving me interesting whiskey and cocktails, or letting me pet their beautiful cats, Anna Waymack and Nate Stetson had a way of keeping my feet firmly planted on the ground. The entire Ithaca Area Ultimate Alliance provided an amazingly fun, energetic, and silly community that, by contrast, tried to get my feet off the ground as much as possible.

When everything locked down in March 2020, leaning on all of the wonderful people listed above felt impossible. Yet, two groups of people made it easier than ever to stay connected. Virtual Argos meetups with Shrutarshi Basu, Eleanor Birrell, Tom Magrino, Marin Cherry, Mae Milano, Laure Thompson, Soumya Basu, Natacha Crooks, Andrew Hirsch, and Isaac Sheff made it feel like everyone was still in Ithaca. Nori Aquino, Jane Brown, Leilani Diaz, John Hawley, Anna Louie, Briana McGeough, Mae Milano, Robert Mustacchi, Nathan Partlan, and Abbie Popa made every other Sunday afternoon feel like a college reunion. I look forward to seeing many of you in person again.

Last but not least, a huge thanks to all of my research colleagues. Thanks to all of my may coauthors over the course of my PhD: Rachit Agarwal, Lorenzo Alvisi, Pedro de Amorim, Owen Arden, Matt Burke, Kyle Croman, Natacha Crooks, Ben Fisch, Sitar Harel, Andrew Hirsch, Ahmed Kosba, Yan Ji, Ari Juels, Ian Miers, Andrew Myers, Haobin Ni, Elaine Shi, Ross Tate, Siqiu Yao, and Fan Zhang. This dissertation would not exist without you. I had the great fortune to be part of two different research groups, both full of brilliant friendly people who were never afraid to be quirky and silly in their own way. Thanks to Josh Acay, Owen Arden, Chin Isradisaikul, Jed Liu, Tom Magrino, Mae Milano, Haobin Ni, Rolph Recto, Silei Ren, Isaac Sheff, Siqiu Yao, Drew Zagieboylo, and Yizhou Zhang in Andrew Myers' Applied Programming Languages group, and Sarah Allen, Iddo Bentov, Kyle Croman, Phil Daian, Steven Goldfeder, Yan Ji, Mahimna Kelkar, Tyler Kell, Sishan Long, Jasleen Malvai, Deepak Maram, Ian Miers, and Fan Zhang in Ari Juels' group. I would not be where I am today if not for all of you.

The work in this dissertation was funded in part by a fellowship awarded through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR), and the Army Research Office (ARO).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

How do we secure highly decentralized computer systems? Many applications today are built from numerous independent communicating services, which often do not trust each other. Despite this lack of trust, the overall system can only function if the services cooperate. Internet of Things sensors and appliances in a smart home must coordinate to be useful, but sensors might be hacked and appliances should still behave reasonably. Miners of blockchain cryptocurrencies like Bitcoin and Ethereum need to agree on the state of the system, yet any of them might try to steal money or censor transactions. Large web applications like Google, Facebook, Amazon, or Twitter are built from numerous independent microservices, yet their code bases are large enough and diverse enough that trusting every service might be dangerous. This combination of structural interconnection and mutual distrust makes it immensely difficult to maintain the integrity of every service.

The integrity of a decentralized system and its data relies on the component services conforming to some system-specific expectations. For example, in a financial transaction processing system, a service recording the movement of money should ensure that all transactions are authorized by their sender and that accounts do not overdraw their balances. A password-based authentication service should ensure that users construct their own password guesses rather than, say, providing a pointer to the location of the password saved in the service's memory.

When all parties are trustworthy, integrity is not a concern, and when all data is public and the system's expectations are made explicit, simply replicating all data and computation, allowing each party to independently verify all transactions, is sufficient to ensure integrity. This approach has become increasingly popular with the proliferation of blockchain systems [e.g., 40, 60, 70, 121, 134, 162, 173]. Unfortunately, many decentralized systems meet neither of these criteria. They must account for potentially untrustworthy actors while including secret data, making implicit assumptions, or both.

This dissertation explores different ways decentralized systems can make explicit the expectations of each constituent module and verify that those requirements are not violated. Chapter 2 introduces Solidus, a protocol built from powerful cryptographic tools that hide secret data while proving that updates conform to an agreed-upon specification. It leverages replication to allow all parties to verify that all updates are declared and contain valid update proofs. Such cryptographic techniques are powerful and flexible, but they have some notable drawbacks. Proving them correct requires careful manual analysis by highly trained experts, and the protocols are often carefully tailored to specific settings and are non-modular, meaning localized changes can destroy the security of the entire system.

Chapters 3 and 4 complement this approach by building language-based tools to provide compositional integrity guarantees. Both chapters study static Information Flow Control (IFC) analysis using type systems. This technique allows developers to specify security policies on code and data and check if those policies are consistent. Previous IFC work either enforces noninterference [72]—a condition that is much too restrictive for real applications—or focuses on confidentiality, defining and eliminating classes of data leakage attacks [44, 96, 103, 120, 145, 177]. Chapters 3 and 4 instead focus on identifying and eliminating classes of integrity attacks even when services must accept data and requests from untrusted sources.

Together this collection of techniques forms a powerful tool set. The cryptographic advances in Solidus show the power of carefully combining cryptographic tools for maintaining both strong confidentiality and integrity in a system based primarily on replication. The advances in IFC capabilities allow for provably-sound compositional analyses of strong, yet expressive, integrity policies in decentralized systems. This combination has the potential to extend and inform a variety of existing and ongoing work to automatically combine IFC and cryptography [4, 135].

1.1 Cryptography and Replication

Ensuring integrity of replicated data is simple when that data is visible to all parties, but most interesting applications contain at least some secret data. Hiding the secret information behind encryptions or cryptographic commitments is a step toward preventing leaks, but it creates two critical challenges for maintaining integrity with replication. First, if not everyone can see the underlying data, replication alone is insufficient for ensuring integrity. Second, it is often not obvious what combination of information an attacker can use to reconstruct secret data.

Both of these shortfalls are becoming increasingly important with the growing prevalence of blockchain systems. Bitcoin, the first major blockchain system, recognized the need for privacy. It uses unique pseudonyms to disconnect the real-world identity of transacting parties from the transactions themselves, and the original paper [121] claims the transactions are anonymous. Yet, the transaction values are still visible, and transactions can be linked by noticing that the same address received coins in transaction T_1 and sent coins in transaction T_2. In 2013, Meiklejohn et al. [108] showed that, with minimal auxiliary information from real-world transactions, this so-called "transaction graph" is sufficient to deanonymize a significant fraction of Bitcoin transactions.

In response to this style of attack, there have been proposals to hide transaction values with various so-called "confidential transaction" protocols [99, 105, 106], or to "mix" transactions together, essentially creating cover traffic for each transaction [77, 104, 112, 141, 166]. These attempts are intuitive and often fairly computationally efficient, but they generally lack rigorous definitions or proofs of security.

Indeed, they generally succeed in maintaining the integrity of the system, but they leak a considerable amount of secret information [114, 158].

Solutions with stronger provable guarantees replicate only encryptions or commitments to values. In doing so, they break the information symmetry between the parties and complicate the task of verifying that each update is valid. Instead, they require cryptographic proofs to demonstrate the integrity of the replicated data. Zerocash [15] and its antecedents [55, 110] provide strong anonymity for Bitcoin-like fully decentralize peer-to-peer payment systems. Hawk [90] provides a compiler for ensuring privacy in smart contracts on Ethereum [173] and similar computational platforms relying on replication for integrity. Though highly effective at ensuring integrity while hiding secret data, these structures are generally inefficient and do not match cleanly with the architecture and needs of existing real-world infrastructure, like the financial system.

Chapter 2 describes Solidus, a protocol with similar goals for confidentiality and integrity, but fitting into a different design point. Solidus takes as a starting point the structure of the existing financial system: a small set of banks each with a large set of accounts. It then builds an anonymous transaction infrastructure on top of this *bank-intermediated* architecture. By moving to a bank-intermediated model, Solidus partially centralizes the payment system, removing the need for novel cryptographic constructs like zk-SNARKs and achieving better performance.

Solidus builds on top of a new cryptographic primitive: the Publicly-Verifiable Oblivious RAM Machine (PVORM). A PVORM provides a means for storing and updating secret data while maintaining integrity in a replication-based system. It completely obscures data and access patterns, yet allows all parties to verify that updates conform to publicly agreed upon requirements (e.g., account balances cannot be negative). In Solidus, each bank maintains its own PVORM of accounts, and transactions link updates between two PVORMs to ensure they correspond. The result is a transaction system that maintains a high level of integrity while leaking almost no information about transactions outside of the banks processing them.

Chapter 2 was previously published as *Solidus: Confidential Distributed Ledger Transactions via PVORM* in the proceedings of the 24th ACM Conference on Computer and Communication Security (CCS '17), co-authored with Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi [37].

1.2 Compositional Integrity Guarantees

Ideally, mechanisms for maintaining integrity in decentralized systems would provide compositional guarantees. That is, demonstrating the security of each module separately would be sufficient to ensure security of the whole system. In a decentralized system, compositional guarantees allow each service to check its security independently, relying only on other services it trusts to properly enforce their security specification. There is no need to rely on *how* those other trusted services maintain their security or on any behavior of untrusted services.

While the Universally Composable (UC) security framework [34] and others it has inspired [33, 171] provide a structure for proving compositionality of cryptographic techniques, using these frameworks is complicated and error-prone. We thus complement the cryptographic techniques discussed above with the language-based approach of Information Flow Control (IFC). IFC tools use labels on data and computation as a security specification, allowing them to track and constrain how data flows through an application. This strategy allows them to enforce entire classes of compositional information security guarantees.

Though IFC tools traditionally use labels to represent confidentiality policies—e.g., Alice may read this data but Bob may not—allowing IFC to prevent unwanted data leaks [66, 159, 175], it can also track and control integrity [19, 178], availability [183], distributed consistency guarantees [111], or combinations thereof [4, 9, 117]. Importantly, IFC guarantees are enforceable with a type system [144] and are highly compositional. Multiple modules can safely compose into a larger application with the same security guarantees as long as the types—including IFC labels—at the module boundaries match and the sensitivity of the code is considered. IFC-based security also performs well in decentralized systems [119], ensuring that you can only be hurt by someone if both you trust them and they fail to enforce their stated security policies.

The gold standard security property for IFC systems is *noninterference* [72]. Noninterference forbids secret (untrusted) data from influencing public (trusted) data in any way. While it will prevent all data leaks or unwanted influence, noninterference is far too constraining for most realistic applications. Many services need to make trustworthy decisions based on input from an untrusted source—*endorsing* that input—or publicly release outputs computed from secret inputs—*declassifying* the information. These operations break noninterference by design and require more nuanced notions of security.

Many IFC systems support developer-specified violations of noninterference through so-called “downgrading operations” like declassification of secrets and endorsement of inputs. The research surrounding these tools defines several different notions of secure declassification [44, 96, 103, 120, 145, 177], but there is little work addressing what it means to safely endorse. Chapters 3 and 4 address this missing piece. They each investigate a different danger of endorsement and how to eliminate corresponding insecurity.

1.2.1 Endorsement With Secrets

Just as in replication-based systems, in the presence of endorsement, secret data complicates maintaining the integrity system. When a service endorses data, it usually implicitly assumes that the provider of that data constructed the input from information it could access. Consider a password checker that takes a guess as input from an untrusted source, compares it to a secret and trusted password value, and then outputs a public and trusted boolean to indicate if the guess was correct. Such a checker clearly assumes the user constructed the password guess using their own data. An attacker should not be able to pass, say, a pointer to the secret password as an argument, allowing them to authenticate despite not knowing the password.

IFC labels provide only a security specification for programs, so this specification should preclude attacks like the one described above. If the confidentiality level of the *guess* is as secret as the stored password secret, the specification would allow an attacker to provide that secret directly despite not being able to read it. If the checker endorsed this input, it would violate the implicit assumption that the attacker constructed the password guess on its own. To enforce this implicit assumption, we compare the confidentiality and integrity of the endorsed data and require that *the source of the endorsed data can read it*.

The idea of connecting confidentiality and integrity to secure downgrading is not new. Robust declassification [120, 177] connects them to define and enforce a notion of secure declassification. Intuitively, it prevents untrusted code or data from influencing declassification decisions. That way attackers cannot cause the wrong data to be declassified or a declassification to occur at the wrong time. The connection, however, was one-sided. Confidentiality security depended on integrity, but not vice versa. The work broke the classic duality between confidentiality and integrity [19] and left open the question of how to secure endorsement.

Chapter 3 deepens the connection between confidentiality and integrity. It formalizes the intuition that endorsing data is only safe when the data's source can read it as *transparent endorsement*, making integrity security depend on confidentiality. Indeed, the definition and enforcement conditions for transparent endorsement are precise mathematical duals to the definition and enforcement conditions of robust declassification, restoring the classic confidentiality–integrity duality.

Robust declassification and transparent endorsement combine to form *nonmalleable information flow* (NMIF), a single unified security condition that applies to programs with an arbitrary mix of declassification and endorsement. Chapter 3 further recasts all three security conditions, framing them as hypersafety properties [46]. In particular, all three are 4-safety properties, meaning they can be defined by sets of four execution traces that do not exhibit certain bad properties—untrusted influence over declassification (robust declassification), secret influence over endorsement (transparent endorsement), or either (NMIF).

Chapter 3 was previously published as *Nonmalleable Information Flow Control* in the proceedings of the 24th ACM Conference on Computer and Communication Security (CCS '17), co-authored with Andrew C. Myers and Owen Arden [36].

1.2.2 Reentrancy

Transparent endorsement and nonmalleable information flow address the assumptions implicit in endorsing data, but many service must also endorse control flow. Normally IFC requires that, to invoke an operation, the caller must be at least as trusted as the operation. For a trustworthy service to accept a request from an untrusted source, however, the service must endorse the control flow, allowing that untrusted source to call a more-trusted request handler. Again, implicit assumptions on the behavior of untrusted parties embedded in the endorsement can endanger system integrity.

Secrets were integral to violating the implicit assumptions of data endorsements, but they are unnecessary to violate the assumptions of control flow endorsements. Most services implicitly assume that requests will execute in a call-and-return pattern where a user makes a request, the service processes the request, and then returns a result. In highly decentralized systems, however, one service might rely on others to process the request, and those other services might be untrustworthy or pass control to someone that is. An attacker can then violate the assumption that the original request will execute in a call-and-return pattern. If it regains control of execution in the middle of processing the first request, the attacker can make a second request, *reentering* the original service.

Without endorsement, standard IFC constraints would only permit these reentrant calls from within the same trust domain as the original service, making it much easier to analyze and control their impacts. Unfortunately, it accomplishes this restriction by preventing *all* requests from parties the service does not trust. For any sort of public-facing application, this prohibition stops nearly all users from submitting requests, rendering the service nearly useless. Instead services will endorse the control flow at the public entry point, allowing anyone to submit requests. Existing IFC techniques provide no restriction on these control flow endorsements, so they fail to detect or prevent reentrancy attacks. As before, a compositional solution is necessary to secure large modular systems.

These reentrant calls can result in serious security vulnerabilities. Attackers have exploited reentrancy bugs in two different Ethereum smart contracts, The DAO [131] and the Uniswap/Lendf.me exchange [127], each time stealing tens of millions of dollars. The attacks have not gone unnoticed, with researchers developing an array of tools and languages to identify reentrancy and other smart contract bugs [e.g., 5, 26, 47, 57, 75, 93, 100]. These approaches universally consider the security of a self-contained object, usually a smart contract. As a result, they aim to secure single contracts—or objects—against reentrancy, resulting in a non-compositional notion of reentrancy that we call *object reentrancy*. A single application made from two smaller objects will have a different definition of reentrancy, and hence security, than a functionally equivalent application written as a single monolithic object. In a decentralized system with different boundaries separating trust domains and objects, these object-based definitions fail to capture the true security concerns of the system.

Chapter 4 recasts the reentrancy attack as a breach of the implicit assumptions of endorsing control flow, as described above. This new lens supports a highly compositional definition: reentrancy occurs when trusted code calls untrusted code that then calls back into trusted code before returning—possibly violating an implicit call-and-return assumption. A reentrancy vulnerability then results when a reentrant call causes an application to violate an invariant it would otherwise maintain, such as incorrectly overdrawing an account balance.

In addition to formalizing these definitions, Chapter 4 describes how locks on integrity levels can provably guarantee security against reentrancy attacks, allowing developers to ignore reentrancy when reasoning about correctness. The locks are enforced with a mix of static and dynamic checking. The static checking allows the type checker to guide developers and avoid the use of expensive runtime checks, while the dynamic portion supports safe and expressive interactions with untrusted code. The result is a powerful, yet expressive tool for maintaining integrity in a decentralized setting with services in different trust domains.

Chapter 4 was previously published as *Compositional Security for Reentrant Ap-*

plications in the proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland '21), co-authored with Siqiu Yao, Haobin Ni, and Andrew C. Myers [39].

CHAPTER 2
SOLIDUS

Blockchain-based cryptocurrencies, such as Bitcoin, allow users to transfer value quickly and pseudonymously on a reliable distributed public ledger. This ability to manage assets privately and authoritatively in a single ledger is appealing in many settings beyond cryptocurrencies. Companies already issue shares on ledgers [58] and financial institutions are exploring ledger-based systems for instantaneous financial settlement.

For many of these companies, confidentiality is a major concern and Bitcoin-type systems are markedly insufficient. Those systems expose transaction values and the pseudonyms of transacting entities, often permitting deanonymization [108]. Concerns over this leakage are driving many financial institutions to restrict on-chain storage to transaction digests, placing details elsewhere [22, 88, 165]. Such architectures discard the key benefits of blockchains as centralized authoritative ledgers and reduce them to little more than a timestamping service.

The overall structure of current blockchains additionally misaligns with that of the modern financial system. The direct peer-to-peer transactions in Bitcoin and similar systems interfere with the customer-service role and know-your-customer regulatory requirements of financial institutions. Instead, the financial industry is exploring a model that we call *bank-intermediated* systems [88, 165]. In such systems a small number of entities—which we call *banks*—manage transactions of on-chain assets on behalf of a large number of users. For example, a handful of retail banks could use a bank-intermediated ledger to authoritatively record stock purchases by millions of customers. By design, bank-intermediated systems faithfully replicate asset flows within modern financial institutions.

While a number of bank-intermediated blockchain systems have been proposed, e.g., [54, 61, 168], these systems either do not provide inherently strong confidentiality or do so by sequestering data off-chain, preventing on-chain settlement. Coin mixes, e.g., [77, 104, 141, 166], and cryptocurrencies such as Monero [112] and Zcash [15] do improve confidentiality, but with notable limitations. Coin mixes and Monero provide only partial confidentiality, with demonstrated weaknesses [108, 114, 158]. Zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [14], on which Zcash is built, provide strong confidentiality. Proof generation, however, is very expensive, requiring over a minute on a consumer machine for Zcash [15]. While this is feasible for a single client performing infrequent transactions, we show experimentally in this paper that adapting zk-SNARKs to a bank-intermediated system would be prohibitively expensive. zk-SNARKs also require an undesirable trusted setup and introduce engineering complexity and cryptographic hardness assumptions that financial institutions are reluctant to embrace [88].

To address these concerns we present *Solidus*,[1] a system supporting strong confidentiality and high transaction rates for bank-intermediated ledgers. Solidus not only conceals transaction values, but also provides the much more technically challenging property of *transaction-graph confidentiality*.[2] This means that a transaction's sender and receiver cannot be publicly identified, even by pseudonyms. They can be identified by their respective banks, but other entities learn only the identities of the banks.

Solidus takes a fundamentally different approach to transaction-graph confidentiality than previous systems such as Zcash. As the technical cornerstone of Solidus, we introduce a new primitive called *Publicly-Verifiable Oblivious RAM Ma-*

[1]The *solidus* was a solid gold coin in the late Roman Empire.

[2]Pseudonymous cryptocurrencies such as Bitcoin are often viewed as graphs where nodes represent keys and edges transactions. The term *transaction-graph confidentiality* means concealing the graph's edges to guard against deanonymization attacks exploiting its structure [108].

chine or *PVORM*, an idea derived from previous work on Oblivious RAM (ORAM). In previously proposed applications, ORAM is used by a single client to outsource storage; only that client needs to verify the integrity of the ORAM. In Solidus, the ORAM stores user account balances. This means that any entity in the system must be able to verify (in zero-knowledge) that bank $\mathcal{B}$'s ORAM reflects precisely the set of valid transactions involving $\mathcal{B}$. To meet this novel requirement, a PVORM defines a set of legal application-specific operations and all updates must be accompanied by ZK proofs of correctness. Correctness includes requirements that account balances remain non-negative, that each transaction updates a single account, and so forth. We offer a formal and general definition of PVORM and describe an implementation incorporated into Solidus.

Introducing the PVORM provides several benefits to Solidus. First, a PVORM can be constructed with either zk-SNARKs or NIZK proofs based on Generalized Schnorr Proofs (GSPs) [29, 32]. GSPs are more efficient to construct than zk-SNARKs and do not require trusted setup, but are much slower to verify, so we explore both options. Second, unlike Zcash, Solidus's core data structure grows only with the number of user accounts, not the number of transactions over the system's lifetime. This property is especially important in high-throughput systems and minimizes performance penalties for injecting of "dummy" transactions to mitigate timing side-channels. Finally, Solidus maintains all balances as ciphertexts on the ledger. This approach supports direct on-chain settlement—a feature many systems, like Zcash, do not aim for. It also permits decryption of balances by authorized parties and allows users to prove their own balances if, for example, they wish to transfer funds away from unresponsive banks.

In addition to the PVORM component, we present a formal security model for Solidus as a whole in the form of an ideal functionality. This presentation may be of independent interest as a specification of the security requirements of bank-intermediated ledger systems. We prove the security of Solidus in this model.

Further, while Solidus targets a permissioned ledger model, it requires only a permissioned group; it is agnostic to the implementation of the underlying ledger, whether centralized or distributed. Therefore, we use the generic term ledger to denote a blockchain substrate that can be instantiated in a wide variety of ways.

Our contributions can be summarized as follows:

- *Bank-intermediated ledgers.* Our work on Solidus represents the first formal treatment of confidentiality on bank-intermediated ledgers—a new architecture that closely aligns with the settlement process in the modern financial system. Our work provides a formal security model that broadly captures the requirements of financial institutions migrating assets onto ledgers.

- *PVORM.* We introduce *Publicly-Verifiable Oblivous RAM Machines*, a new construction derived from ORAM and suitable for enforcing transaction-graph confidentiality in ledger systems. We offer formal definitions and efficient constructions using Generalized Schnorr Proofs.

- *Implementation and Experiments.* We report on our prototype implementation of Solidus and present results of benchmarking expreiments, demonstrating a lower bound on Solidus performence. We also provide a performance comparison with zk-SNARKs.

Our results are not just a new technical approach to transaction-graph confidentiality on ledgers. They also show the practicality of bank-intermediated ledger systems with full on-chain settlement.

2.1 Background

We now review existing cryptocurrency schemes and approaches to their confidentiality. We then give some background on bank-intermediated system modeling and describe the technical building blocks used to achieve security and confidentiality in Solidus.

2.1.1 Existing Cryptocurrencies

Many popular cryptocurrencies are based on the same general transaction mechanism popularized by Bitcoin. Any user $\mathcal{U}$ may create an account ("address" in Bitcoin) with a public/private key pair. To transfer money, $\mathcal{U}$ creates a transaction T by signing a request to send some quantity of coins to a recipient.[3] Miners sequence transactions and directly publish T to the *blockchain*, an authoritative append-only record of transactions. Since only transactions are recorded, to determine the balance of $\mathcal{U}$, it is necessary to tally all transactions involving $\mathcal{U}$ in the entire blockchain. As a performance optimization, many entities maintain a balance summary—called an unspent transaction (UTXO) set in Bitcoin.

This setup publicizes all account balances and transaction details. The only confidentiality stems from the pseudonymity of public keys which are difficult—though far from impossible [108]—to link to real-world identities.

To conceal balances and transaction values, Maxwell proposed a scheme called Confidential Transactions (CT) [105]. CT operates in a Bitcoin-like model, but publishes only Pedersen commitments of balances. Transaction values are similarly hidden and balances are updated using a homomorphism of the commitments

[3]This is a simplification and details vary between systems. For example, a basic transaction in Bitcoin ("Pay-to-PubkeyHash"), takes a reference to the output from a previous transaction and includes a small script restricting the user of outputs and a mining fee.

and proven non-negative using Generalized Schnorr Proofs (see below). Solidus uses an El-Gamal-based variant of CT to conceal transaction values.

Several decentralized cryptocurrency schemes aim to provide partial or full transaction-graph confidentiality. (See Section 2.7 for a brief overview.) As noted above, though, only those involving zk-SNARKs provide strong confidentiality of the type we seek for Solidus. Zcash and offshoots such as Hawk [90], for example, conceal balances, transfer amounts, and the transaction graph. They do not, however, aim to align with the financial settlement system. Additionally, they require trusted setup and store authoritative state in a Merkle tree that grows linearly with the total system transaction history, drawbacks we avoid in the design of Solidus. As a basis for performance comparison, we describe and evaluate a zk-SNARK-based version of Solidus in Section 2.6.3.

2.1.2 Bank-Intermediated Systems

Managing assets on ledgers is appealing to the financial industry.

Asset transfers in financial markets today involves a laborious three-step process. *Execution* denotes a legally enforceable agreement between buyer and seller to swap assets, such as a security for cash. *Clearing* is updating a ledger to reflect the transaction results. *Settlement* denotes the exchange of assets after clearing. Multiple financial institutions typically act as intermediaries; when a customer buys a security, a broker or bank will clear and settle on her behalf via a clearinghouse.

Today, the full settlement process typically takes three days (T+3) for securities. This delay introduces systemic risk into the financial sector. Government agencies such as the Securities and Exchange Commission (SEC) are trying to reduce this delay and are looking to distributed ledgers as a long-term option. If asset titles—the authoritative record of ownership—are represented on a ledger, then trades could execute, clear, and settle nearly instantaneously.

Existing cryptocurrencies such as Bitcoin can be viewed as titles of a digital asset. Execution takes the form of digitally signed transaction requests, while clearing and settlement are simultaneously accomplished when a block containing the transaction is mined[4]

Today, however, banks intermediate most financial transactions. Even with Bitcoin, many customers defer account management to exchanges (e.g. Coinbase). Additionally, a labyrinthine set of regulations, such as Know-Your-Customer [116], favors bank-intermediated systems. Thus existing cryptocurrencies do not align well with either financial industry or ordinary customer needs.

Solidus aims to provide fast transaction settlement in a bank-intermediated ledger-based setting. As in standard cryptocurrencies, Solidus assumes that each user has a public/private key pair and digitally signs transactions. Solidus, however, conceals account balances and transaction amounts as ciphertexts. To do so and provide public verifiability at the same time, it relies on PVORM.

2.1.3 Oblivious RAM

As PVORM is heavily inspired by *Oblivious RAM* (ORAM), we provide some background here.

An ORAM is a cryptographic protocol that permits a client to safely store data on an untrusted server. The client maintains a map from logical memory addresses to remote physical addresses and performs reads and writes remotely. Ensuring freshness, integrity, and confidentiality of data in such a setting is straightforward using authenticated encryption and minimal local state. The key property of ORAM is *concealment of memory access patterns*; a polynomially-bounded adver-

[4]Strictly speaking, settlement involves an exchange of assets, and thus two transactions, but this issue lies outside the scope of our work.

sarial server cannot distinguish between two identical-length sequences of client operations.

These properties provide an appealing building block for Solidus. Identifying an edge in the system's transaction graph can easily be reduced to identifying which account's balance changed with a transaction. Thus placing all balances in an ORAM immediately provides transaction graph confidentiality. Moreover, recent work has drastically improved the performance of ORAM. The most practical ORAM constructions maintain a small local cache on the client known as a *stash* and either organize the data blocks as a tree allowing logarithmic work on each access [160, 169], or write to completely randomized locations, resulting in constant-time writes but linear reads (so-called "write-only" ORAM) [21, 137].

Unfortunately, standard ORAM is insufficient for Solidus. Because ORAM is designed for a client using an untrusted server, correctness simply means the ORAM reflects the client's updates. There is no notion of "valid" updates, let alone means for a client to prove an update's validity. In Solidus, clients (banks) must prove an application-defined notion of correctness for each update. Banks also cannot store a local stash, as we would no longer have all data on the ledger. To address these concerns we introduce PVORM—detailed in Section 2.3—a new construction inspired by ORAM.

2.1.4 Generalized Schnorr Proofs

Solidus makes intensive use of *Generalized Schnorr Proofs* (GSPs), a class of Σ-protocol for which practical honest-verifier zero-knowledge arguments (or proofs) of knowledge can be constructed.

Notation introduced in [29, 32] offers a powerful specification language for GSPs that call the PoK language. Using multiplicative group notation, let $G = \langle g \rangle$

be a cyclic group of prime order p.[5] If $x \in \mathbb{Z}_p$ and $y = g^x$, then $\mathsf{PoK}(x : y = g^x)$ represents a ZK proof of knowledge of x such that $y = g^x$ where g and y are known to the verifier. (This is the Schnorr identification protocol.)

The PoK specification language for GSPs is quite rich; it supports arbitrary numbers of variables as well as conjunctions and disjunctions among predicates. It has a set of corresponding standard tools based on the Schnorr identification protocol for efficient realization in practice when G has known order [32]. It is possible, additionally, using the Fiat-Shamir heuristic [67], to render GSPs non-interactive, i.e., to generate NIZK proofs of knowledge.

Solidus uses GSPs in a variety of ways to ensure account balances and PVORMs are properly updated and remain valid.

2.2 Solidus Overview

Before delving into technical details, we give an overview of Solidus, including basic notation, trust assumptions, and security goals. We also give an architectural sketch. First, however, we give a concrete target application as motivation.

Example 2.1 (*TradeWind Markets*)**.** TradeWind Markets, whose use case helped inform the design of Solidus, offers an example of how Solidus might support management of asset titles [165]. TradeWind is building an electronic communication network (ECN) for physical gold bullion to be traded using a bank-intermediated ledger for settlement and title management. The physical bullion is managed by a custodian who is trusted to track inflows and outflows to and from a specifically designated vault. Each user has an account with a *holding bank*—generally a large commercial bank—which manages trades. A user may additionally buy gold from

[5]Solidus uses the group for elliptic curve secp256k1. We make this choice for performance, so despite elliptic curve groups typically using additive notation, we will use multiplicative notation for simplicity and generality.

outside, send it to the vault, and sell it on the TradeWind ECN—requiring the custodian to create a record of the asset—or buy gold on the TradeWind ECN, remove it from the vault, and sell it elsewhere—requring the custodian to destroy the asset record.

Holdings are represented on the ledger as fractional ounces of gold held by individual users. To trade gold, a user authorizes her holding bank to transfer the gold to another user. Holding banks may also provide other services, such as holding gold as collateral against a loan. In such cases the bank may freeze assets, for example, until the loan is repaid.

As we shall show, Solidus can support the full asset lifecycle of a system like the TradeWind ECN while providing practical performance and strong confidentiality and verifiability guarantees.

2.2.1 Design Approach

Solidus has two important features that differ from existing ledger systems and make it more amenable to the financial industry.

The first is its *bank-intermediated* design: unlike nearly all systems proposed by the research community (see Section 2.7), Solidus aligns with the structure of the modern financial system. Each bank in Solidus has a set of customers or *users* who hold shares of some asset (e.g., securities, cryptocurrency, or gold) in their accounts. Specially designated entities called *asset notaries* record the injection of assets into the system, as we discuss below. Second, unlike other bank-intermediated systems, Solidus provides *strong confidentiality*. It conceals account balances and transaction details from non-transacting entities, placing them on the ledger as ciphertexts. It is for these reasons that Solidus uses PVORM. Each bank maintains its own PVORM on the ledger to record the identities and balances of its account.

Each transaction involves a sending user at a sending bank, and a receiving user at a receiving bank. When a user (sender) $\mathcal{U}_s$ signs a transaction and gives it to her (sending) bank $\mathcal{B}_s$, $\mathcal{B}_s$ first verifies the validity of the transaction—that it is correctly signed and $\mathcal{U}_s$ possesses the funds $\$v$ to be sent—then updates its PVORM to reflect the results of the transaction. The receiving bank performs a corresponding update on the receiving user's account.

The confidentiality properties of PVORM ensure that another entity can learn only the identities of the sending and receiving banks, not $\$v$ or the identities of the transacting users. Indeed, even the sending bank cannot identify the receiving user nor the receiving bank the sending user.[6] The public verifiability of PVORM ensures that any entity with access to the ledger can verify that each transaction is correctly processed by both banks.

Solidus's design is agnostic to the implementation of the underlying ledger. While it does require a mutually-aware group of banks and transaction validation by the ledger maintainers, those maintainers can be a "permissioned" (fixed-entity) group, an "unpermissioned" (fully decentralized) ledger (a blockchain), or any other trustworthy append-only data structure.

2.2.2 Architectural Model

In Solidus, a predetermined set of banks $\mathcal{B}_1, \ldots, \mathcal{B}_m$ maintain asset titles on a ledger. Each bank $\mathcal{B}_i$ has a public/private key pair for each of encrypting and signing. It also has up to n users $\{\mathcal{U}^i_j\}^n_{j=1}$ each with a signature key pair. Each account is publicly associated with one bank, so $\text{bank}(\mathcal{U}^i_j) = \mathcal{B}_i$ is well-defined.

Each bank $\mathcal{B}_i$ maintains its own private data structure M_i containing each user's balance and public key. It maintains a corresponding public data structure C_i,

[6]It is desirable for receiver to be able to verify the sender's identity. The sender can easily acquire a receipt by retaining a proof that she authorized the transaction.

placed on the ledger, whose elements are encrypted under $\mathcal{B}_i$'s encryption key. M_i and C_i together constitute the memory in a PVORM, which we describe in Section 2.3. Solidus uses this structure to ensure that updates to C_i reflect valid transactions processed in M_i while concealing transaction details and the transaction graph.

A transaction T is a digitally signed request by user $\mathcal{U}_j^i$ with balance $\$b_j^i$ to send some amount $\$v$ of asset to another user $\mathcal{U}_{j'}^{i'}$. The transaction is valid if $\$b_j^i \geq \$v \geq 0$. To process a transaction, $\mathcal{B}_i$ updates M_i to set $\$b_j^i \leftarrow \$b_j^i - \$v$ and $\mathcal{B}_{i'}$ updates $M_{i'}$ to set $\$b_{j'}^{i'} \leftarrow \$b_{j'}^{i'} + \$v$. They generate publicly verifiable ZK-proofs that $\$v \geq 0$ and that they updated their respective PVORMs correctly using $\$v$. Figure 2.1 depicts a simple Solidus transaction.

We treat the ledger as a public append-only memory which verifies transactions. All banks have asynchronous authenticated read and write access and the ledger accepts only well-formed transactions not already present. We model this by an ideal functionality $\mathcal{F}_{\text{Ledger}}$, detailed in Section 2.4, which any bank can invoke.

Notarizing New Asset Titles. As described above, all user transactions must be zero-sum; $\mathcal{U}_j^i$ sends money (that she must have) to $\mathcal{U}_{j'}^{i'}$. Financial systems, however, are generally not closed; assets can enter and leave the system through specific channels. To support this, Solidus defines a fixed set of *asset notaries* $\{\mathcal{U}_1^\$, \ldots, \mathcal{U}_\ell^\$\}$. These are accounts with no recorded balance, but the authority to create and destroy asset titles. To ease auditing of this sensitive action, transactions involving $\mathcal{U}_i^\$$ reveal its identity.

Asset notaries clearly must be restricted; it would make no sense to allow arbitrary users to create and destroy asset titles. In Example 2.1, Solidus would designate the custodian as the sole notary responsible for acknowledging receipt and removal of the physical asset (gold) and guaranteeing its physical integrity.

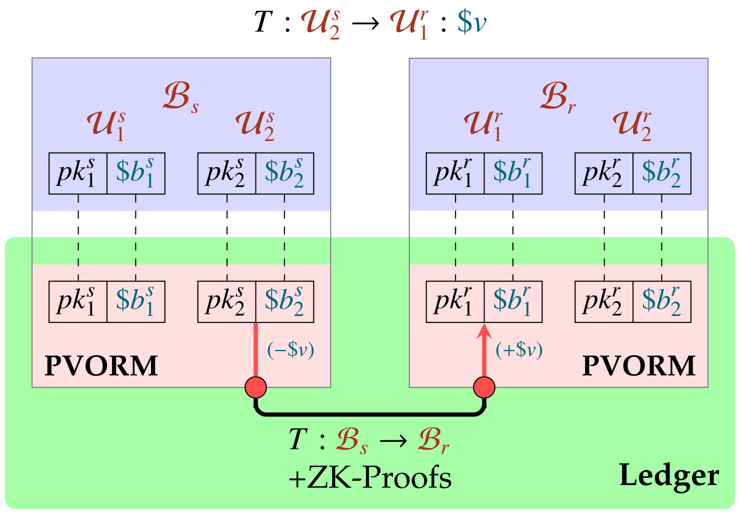


Figure 2.1: An example transaction T where $\mathcal{U}_2^s$ at $\mathcal{B}_s$ sends $\$v$ to $\mathcal{U}_1^r$ at $\mathcal{B}_r$ and each bank has two users. The upper boxes are the logical (plaintext) memory of each bank's PVORM, and the lower boxes are the associated public (encrypted) memories. Entities other than $\mathcal{B}_s$, $\mathcal{B}_r$, $\mathcal{U}_2^s$, and $\mathcal{U}_1^r$ learn only that a user at $\mathcal{B}_s$ sent money to a user at $\mathcal{B}_r$ and both banks updated their PVORMs correctly.

2.2.3 Trust Model

Solidus assumes that banks respect the confidentiality of their own users but otherwise need not behave honestly. They may attempt to steal money, illicitly give money to others, manipulate account balances, falsify proofs, etc. Banks (respectively, users) can also attempt to violate the confidentiality of other banks' users (respectively, other users) passively or actively. We assume no bound on the number of corrupted banks or users, which may collude freely.

The Ledger. We assume the ledger abstraction given in Section 2.2.2. In practice, the ledger can, but need not, be maintained by the banks themselves. If not maintained by the banks, the ledger's trust model is independent from the higher-level protocol. It may be constructed using a (crash-tolerant) consensus protocol such as Paxos [94], ZooKeeper [82], or Raft [124], a Byzantine consensus protocol such as PBFT [35], a decentralized consensus protocol such as Nakamoto consensus [121], or even a single trustworthy entity. We simply assume that the ledger maintainers satisfy the protocol's requirements and the ledger remains correct and available.

We regard the ledger together with the public PVORM data structures $\{C_i\}$ as a replicated state machine. Despite this, Solidus's flexible design allows us to treat the consensus and application layers as entirely separate for the majority of our discussion.

Availability. We assume that the ledger remains available at all times; it is not susceptible to denial-of-service attacks and enough consensus nodes will remain non-faulty to maintain liveness. A bank, however, can be unavailable in two ways: it can freeze a user's assets by rejecting transactions or it can go offline entirely.

Asset freezing can be a feature. For certain types of assets (e.g. gold, as in Example 2.1) a user may wish to use her balance as collateral against a loan. A bank could, however, maliciously freeze a user's assets or go offline due to a technical or business failure. In either case, an auditor with the bank's decryption key (see below) could enable a user to prove her balance and recover funds despite being unable to transact directly.

Auditing. Regulators and auditors play a pivotal role in the financial sector. While Solidus does not include explicit audit support, it enables banks to prove correct decryption of on-chain data or share their private decryption key. In the first case, the auditor can acquire a transaction log on demand and verify its correctness and completeness. In the second case, the auditor can directly and proactively monitor activity within the bank and its accounts.

Network. We do not assume a network adversary. An active network adversary would make the availability requirement of the ledger impossible, while a passive adversary can be mostly mitigated simply by securing all messages with TLS. The existence of communication between users and their banks could still leak information, but this is inherent in any bank-intermediated system and could be mitigated using Tor [62] or similar protocols.

2.2.4 Security Goals

Solidus aims to provide very strong safety and confidentiality guarantees for both individual users and the system as a whole.

Safety Guarantees. Solidus provides a very simple but strong set of safety guarantees. First, no user's balance may decrease without explicit permission of that user (in the form of a signature), and such authorization can be used only once; there are no replay attacks. Second, account balances can never be negative, ensuring that no user can spend money she does not have. Finally, transactions that do not include asset notaries must be zero-sum.

To ensure the above properties hold, we require that the correctness of every transaction be proved in a publicly-verifiable fashion (via ZK-Proof). If the ledger checks these proofs before accepting—and settling—the transaction, then every transaction will maintain these guarantees. Solidus places all proofs on the ledger, meaning an auditor can verify them offline.

Maintaining these guarantees requires all transactions involving a single bank to be serialized. Banks can use the serialization provided by the ledger or another locking mechanism to accomplish this, but everyone must agree on the ordering.

Confidentiality Guarantees. To facilitate audits and asset recovery against malicious banks, Solidus places all account balances and transaction details directly on the ledger. Despite this persistent public record, Solidus provides a strong confidentiality for all users. First, account balances are visible only to the user's bank (and authorized auditors). Second, while transactions do reveal the sending and receiving banks, there is no way to determine if two transactions involving the same bank involved the same account. We use a hidden-public-key signature scheme (see Section 2.8.3) to enforce the publicly-verifiable authorization requirement above without revealing identities. This second feature is often referred to as *transaction graph confidentiality*. It precludes use of the pseudonymous schemes employed by Bitcoin and similar systems, and is the challenge specifically addressed by PVORM.

We do not directly address information leaked by the timing of transactions. These channels and the bank-level transaction graph can, however, be eliminated by requiring each bank to post transactions at regular intervals in batches of uniform size. These batches would be padded out by "dummy" transactions of value 0 to obscure which banks conducted real transactions.

We present a formal model in Section 2.4 that encompasses all of these security and confidentiality goals.

2.3 PVORM

As discussed in Section 2.1.3, ORAM presents a means to conceal the Solidus transaction graph, but lacks the public verifiability that Solidus requires. To overcome this limitation, we introduce the *Publicly-Verifiable Oblivious RAM Machine* (PVORM).

As with ORAMs, PVORMs have a private logical memory M and corresponding encrypted physical memory C. There are, however, four key differences:

- *Constrained Updates.* Write operations are constrained by a public function f. In Solidus, for example, M contains account IDs and balances and f updates a single balance to a non-negative value.
- *Publicly Verifiable Updates.* Whenever the client modifies C, it must publicly

prove (in zero-knowledge) that the change reflects a valid application of f.

- *Client Maintains All Memory.* Instead of a client maintaining M and a server maintaining C, the client maintains both directly. While M remains hidden, C is now publicly visible (e.g., on a ledger in Solidus).
- *No Private Stash.* Any data in M not represented in C would prevent the client from proving correctness of writes. Instead of a variable-size private stash, PVORM includes a fixed-size public encrypted stash.

To achieve public verifiability, our PVORM construction relies on public-key cryptography. Another example of an ORAM scheme that uses public key cryptography is Group ORAM [101], which does so for a more standard cloud setting, rather than our setting here. In fact, while traditional ORAMs generally uses symmetric-key primitives, this difference is not fundamental. One could construct a PVORM using symmetric-key encryption and zk-SNARKs, but as we see in Section 2.6.3, such a construction performs poorly.

We also leverage the fact that PVORM is designed for public verifiability and not storage outsourcing to improve efficiency. In ORAM, reads incur a cost as the client must retrieve data from the server. In PVORM, reads are "free" in that they require only reading public state—the ledger in Solidus—which leaks nothing. Writes, however, are still publicly visible. Second, since PVORM does not aim to reduce local memory usage, we assume that the client locally maintains a full copy of the PVORM including private data and metadata. This allows clients to perform updates much more efficiently by avoiding unnecessary decryption.

These features are nearly identical to those leveraged by write-only ORAM, but those techniques do not apply. Write-only ORAM requires simple writes, but we implement updates as read-update-write operations to prove properties about changes in values.

2.3.1 Formal Definition

We now present a formal definition of PVORM. We let M represent a private array of values from a publicly-defined space (e.g. $\mathbb{N}$) and C be the public (encrypted) representation of M. U is the space of update specifications (e.g., account ID, balance change pairs).

Definition and Correctness. We first define the public interface of a PVORM and its correct operation. A PVORM consists of the following operations.

- $\mathsf{Init}(1^\lambda, n, m_0, U) \xrightarrow{\$} (pk, sk, C)$, a randomized function that initializes the PVORM with security parameter 1^λ, n data elements, initial memory $M = (m_0, \ldots, m_0)$, and valid update values U.
- An update constraint function $f(u, M) \rightarrow M'$ that updates M according to update $u \in U$. Note that f may be undefined on some inputs (invalid updates), and must be undefined if $u \notin U$.
- $\mathsf{Update}(sk, u, C) \xrightarrow{\$} (C', e, \mathit{proof})$, a randomized update function that takes an update u and a public memory and emits a new public memory, a ciphertext e of u, and a zero-knowledge proof of correct application.
- $\mathsf{Ver}(pk, C, C', e, \mathit{proof}) \rightarrow \{\mathsf{true}, \mathsf{false}\}$, a deterministic update verification function.

We also define $\mathsf{Read}(sk, C) \rightarrow M$ and $\mathsf{Dec}(sk, e) \rightarrow u$, two deterministic functions that read every value from a C as a plaintext memory M and decrypt an update ciphertext, respectively. We employ these operations only in our correctness and security definitions; they are not part of the core PVORM interface.

We define correctness of a PVORM with respect to valid update sequences. An update sequence $\{u_0\}_{i=1}^k$ is *valid for* m_0 if, when we let $M_0 = (m_0, \ldots, m_0)$ and $M_i = f(u_i, M_{i-1})$, then M_i is defined for all $0 \leq i \leq k$. A PVORM is *correct* if for all initial values m_0 and all update sequences $\{u_i\}_{i=1}^k$ valid for m_0,

$$\Pr[\mathbf{Exp}^{\text{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)] = 1$$

where $\mathbf{Exp}^{\text{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)$ is defined as

```
Experiment $\mathbf{Exp}^{\text{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)$:
  $(pk, sk, C_0) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, m_0, U)$
  if $\mathsf{Read}(sk, C_0) \neq M_0$, return false
  for $i = 1$ to $k$ :
    $(C_i, e_i, proof_i) \xleftarrow{\$} \mathsf{Update}(sk, u_i, C_{i-1})$
    if $[(\mathsf{Read}(sk, C_i) \neq M_i) \vee (\mathsf{Dec}(sk, e_i) \neq u_i)$
        $\vee \neg\mathsf{Ver}(pk, C_{i-1}, C_i, e_i, proof_i)]$
      return false
  return true
```

with $\{M_0, \ldots, M_k\}$ defined as above. In other words, the PVORM is correct if Update correctly transforms C as defined by f and Ver verifies these updates.

Obliviousness. Solidus requires a structure that can realize ORAM guarantees in a new setting against even an adaptive adversary. Intuitively, we require the PVORM to guarantee that any two adaptively-chosen valid update sequences result indistinguishable output. Formally, we say that a PVORM is *oblivious* if for all PPT adversaries $\mathcal{A}$, there is a negligible $negl(\lambda)$ such that for all $n \in \mathbb{N}$, m_0, and U,

$$\Big| \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(0, \mathcal{A}, \lambda, n, m_0, U) = 1\Big] - \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(1, \mathcal{A}, \lambda, n, m_0, U) = 1\Big]\Big| \leq negl(\lambda)$$

where $\mathbf{Exp}^{\text{Obliv}}(b, \mathcal{A}, \lambda, n, m_0, U)$ is defined by

```
Experiment $\mathbf{Exp}^{\text{Obliv}}(b, \mathcal{A}, \lambda, n, m_0, U)$:
  $(pk, sk, C) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, m_0, U)$
  return $\mathcal{A}^{O_{b,sk,C}(\cdot,\cdot)}(1^\lambda, pk, C)$
```

where $O_{b,sk,C}(\cdot,\cdot)$ is a stateful oracle with initial state $S \leftarrow C$. On input (u_0, u_1), $O_{b,sk,C}$ executes $(C', e, \mathit{proof}) \xleftarrow{\$} \mathsf{Update}(sk, u_b, S)$, updates $S \leftarrow C'$, and returns (C', e, proof). The experiment aborts if any C' is ever undefined.

This definition is an adaptive version of those presented in the ORAM literature [153, 160, 169].

Public Verifiability. The final piece of our security definition is that of public verifiability. Intuitively, we require that each update produce a proof that the update performed was valid and is the one claimed. Formally, a PVORM is *publicly verifiable* if for all PPT adversaries $\mathcal{A}$,

$$\Pr[\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)] \leq \mathit{negl}(\lambda)$$

where $\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)$ is defined as

```
Experiment $\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)$:
  $(pk, sk, \_) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, \_, \_)$;
  $(C, C', e, \mathit{proof}) \xleftarrow{\$} \mathcal{A}(1^\lambda, n, pk, sk)$;
  return $\mathsf{Ver}(pk, C, C', e, \mathit{proof})$
    $\wedge\ (f(\mathsf{Dec}(sk, e), \mathsf{Read}(sk, C)) \neq \mathsf{Read}(sk, C'))$
```

This corresponds to the soundness of the ZK-proof that an update was performed correctly.

2.3.2 Solidus Instantiation

In Solidus we instantiate a PVORM by combining several GSPs with the structure of Circuit ORAM [169]. Circuit ORAM places data blocks into buckets organized as a binary tree. It performs updates by swapping pairs of blocks along paths in that tree. This structure leads to good performance for two reasons: updates require logarithmic work in the number of accounts, and pairwise swaps of public-key

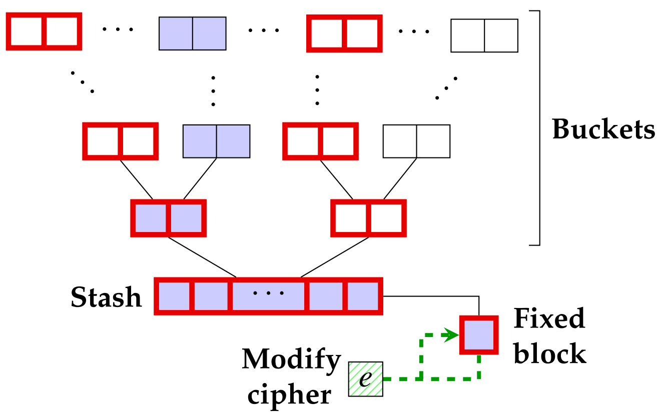


Figure 2.2: An update for a Circuit ORAM-based PVORM with buckets of size 2. Colors indicate the blocks involved in each operation of the read-modify-write structure. Read moves one block from the read path (shaded) into the distinguished fixed block. Then modify combines it (homomorphically) with the modify ciphertext (dashed). Finally write evicts the resulting value into the tree along two eviction paths (thick bordered).

ciphertext admit efficient ZK-proofs of correctness. Figure 2.2 shows how Solidus's PVORM is structured and updated.

Each data block holds an account's unique identifier and balance. This pair of values must move in tandem as blocks are shuffled, so Solidus employs a verifiable swap algorithm for El Gamal ciphertexts [85] augmented to swap ordered pairs of ciphertexts (see Section 2.8.4).

Solidus constrains each update to modify one account balance and requires that balances remain in a fixed range $[0, N]$. To make updates publicly verifiable, a bank first moves the desired account to a deterministic fixed block by swapping that position with each block along the Circuit ORAM access path. Next the bank updates the account balance and generates a set inclusion proof on the resulting ciphertext to prove it is in the legal range (see Section 2.8.5). Finally, the bank performs Circuit ORAM's eviction algorithm to reinsert the updated account. This again requires swapping the fixed block with a set of tree paths.

In Section 2.9 we concretize this construction and prove that it is correct, oblivious, and publicly verifiable.

Stash Overflow. Circuit ORAM assumes a stash of bounded size, but data loss is possible if the stash overflows, resulting in a probabilistic definition of correctness; correct behavior occurs only when data is not lost. Since the probability of data loss is negligible in the size of the stash, the definition is reasonable for the setting.

In Solidus the stash must be placed on the ledger, so to prevent leaking information we also bound the stash size. Data loss is, however, catastrophic no matter how infrequent. When the stash would overflow, instead of losing data we insert one account deeper into the tree. This insertion is public, so it does leak that regular eviction was insufficient as well as the location of a single account (though not the account's identity).

Solidus inherits the stash overflow probability of Circuit ORAM, which is negligible in the stash size [169]. As we show in Section 2.6, the PVORM update performance is linear in the stash size, giving Solidus a direct performance-privacy trade-off. Pleasantly, modest stash sizes make overflow exceedingly unlikely. With buckets of size 3, a stash of size 25 reduces overflow probability to near 2^{-64}.

2.4 Solidus Protocol

We now present the Solidus protocol. This construction relies heavily on cryptographic primitives that we describe in Section 2.8. We make this choice to simplify the explanation and leave abstract operations with several instantiations—such as range proofs.

Bank State. The state of a bank $\mathcal{B}_i$ consists of an encryption key pair (ePK_i, eSK_i), a signing key pair (sPK_i, sSK_i), and a set of accounts. Each account $\mathcal{U}_j$ has a unique account identifier and a balance. For simplicity, we use $\mathcal{U}_j$'s public key pk_j as its identifier.

Each bank maintains its own PVORM, updated on every transaction, containing the information of each of its accounts. Section 2.3.2 describes the PVORM structure.

Requesting Transactions. As Solidus is bank-intermediated, $\mathcal{U}_s$ at $\mathcal{B}_s$ must send a request to $\mathcal{B}_s$ in order to send $\$v$ to $\mathcal{U}_r$ at $\mathcal{B}_r$. The request consists of:

- A unique ID txid
- $\mathsf{Enc}(ePK_s, \$v)$, $\$v$ encrypted under $\mathcal{B}_s$'s public key
- $\mathsf{Enc}(ePK_r, pk_r)$, a ciphertext of $\mathcal{U}_r$'s ID under $\mathcal{B}_r$'s public key
- A hidden-public-key signature signed with sk_s (see Section 2.8.3).

On receipt of a request, $\mathcal{B}_s$ must validate the request—check that txid is globally unique and $0 \leq \$v \leq \$b_s$—and initiate the transaction settlement process.

Settling Transactions. Figure 2.3 shows the structure of settling a transaction in Solidus. $\mathcal{B}_s$ generates a proof that $\$v \geq 0$, reencrypts $\$v$ under ePK_r, and sends $(\mathsf{txid}, \mathsf{Enc}(ePK_r, \$v), \mathsf{Enc}(ePK_r, pk_r))$ to $\mathcal{B}_r$. Then both banks (concurrently) update their respective PVORMs, sign their updates, and post all associated proofs and signatures onto the ledger. Once the full transaction is accepted by the ledger, the assets have been transferred and the transaction has settled.

Transaction IDs. To prevent replay attacks, Solidus includes a globally unique ID with each transaction. This ID could simply be a random bit string (eg., a GUID), but then verification would require the ID of every transaction over the lifetime of the system. To avoid this growing cost, Solidus uses a two-part transaction ID: a timestamp and a random number. Transactions are only valid within a time window T_Δ. If $\mathsf{txid} = (T, \mathsf{id})$, the transaction is only valid at time T_{now} if

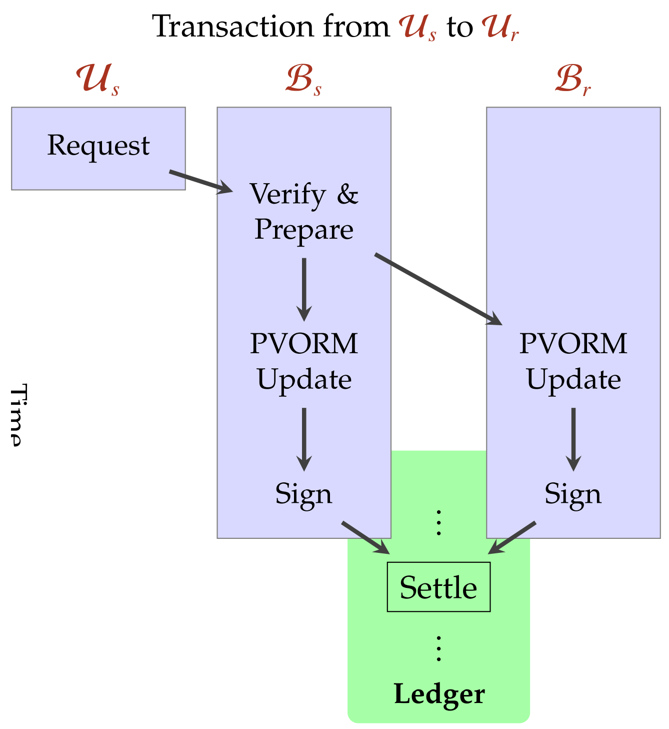


Figure 2.3: The lifecycle of a transaction in Solidus. An arrow from one operation to another means the second depends on the first. Note that $\mathcal{U}_r$ does not appear. The receiving user plays no role in settling transactions.

$T_{\text{now}} - T_\Delta < T < T_{\text{now}}$. This allows verification to only store IDs for T_Δ and still properly prevent replay attacks.

Opening and Closing Accounts. Banks are constantly opening new accounts, so Solidus must support this. To create an account, bank $\mathcal{B}_i$ must insert the account into its PVORM. Our construction makes this simple. $\mathcal{B}_i$ publishes the new ID with a verifiable encryption of the ID and balance 0. It then inserts this ciphertext pair into its PVORM by replacing a dummy value. To close an account $\mathcal{B}_i$ simply publicly verifies the identity of an account and replaces it in the PVORM with a dummy value.

```
$\mathcal{F}_{\text{Init}}\left[\lambda, \{\mathcal{B}_i\}_{i=1}^k, \{\mathcal{U}_i\}_{i=1}^n\right]$

Init:
  for $i \in [1, n]$:
    Generate key pair $(pk_i, sk_i) \xleftarrow{\$} \mathsf{hGen}(1^\lambda)$
    send $pk_i$ to each user and bank and $(pk_i, sk_k)$ to $\mathcal{U}_i$
  for $i \in [0, k]$:
    Generate key pair $(sPK_i, sSK_i) \xleftarrow{\$} \mathsf{sGen}(1^\lambda)$
    $(ePK_i, eSK_i, C_i) \xleftarrow{\$} \mathsf{Init}(1^\lambda, |\mathcal{B}_i|, 0, U)$
    send ("initBank", $ePK_i, sPK_i, C_i$) to each user and bank
    send all five values to $\mathcal{B}_i$
```

Figure 2.4: Solidus ideal initializer with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$.

2.4.1 $\mathcal{F}_{\text{Ledger}}$-Hybrid Functionality

For simplicity we define the Solidus protocol, $\mathbf{Prot}_{\text{Sol}}$, using a trust initializer and an idealized ledger. We could instantiate the trusted initializer using existing PKI systems and, as mentioned above, Solidus is agnostic to the ledger implementation so we wish to leave that abstract. We present the trusted initializer $\mathcal{F}_{\text{Init}}$ in Figure 2.4 and the ledger $\mathcal{F}_{\text{Ledger}}$ in Figure 2.5. Throughout the protocol, users employ hidden-public-key signatures (see Section 2.8.3) and banks employ Schnorr signatures [32, 148], denoted $(\mathsf{sGen}, \mathsf{Sign}, \mathsf{sVer})$.

The $\mathcal{F}_{\text{Ledger}}$ functionality has two operations: posting a completed transaction and aborting an in-progress transaction. The need for the first is obvious; the ledger is the authoritative record of transactions and is responsible for their verification. The second helps guard against malicious activity. As we see below, processing a transaction from $\mathcal{U}_s$ requires bank $\mathcal{B}_s$ to send its PVORM update to $\mathcal{B}_r$ prior to posting the transaction to the ledger, but $\mathcal{B}_r$ may never reply. With no abort operation, $\mathcal{B}_s$ has two options: wait for a reply—causing a DoS attack if none arrives—or proceed as if the transaction were never initiated. In the second case, $\mathcal{B}_r$ can learn with high probability whether $\mathcal{U}_s$ participates in future transactions involving $\mathcal{B}_s$; if a different Circuit ORAM path is accessed, $\mathcal{U}_s$ is not involved, but if the same

```
$\mathcal{F}_{\text{Ledger}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}\right]$

Init: TXID = $\emptyset$ and Ledger = $\epsilon$

On receive ("aprvRecvTxn", txid, txn):
    assert txid $\notin$ TXID
    Parse txn $\to (\mathcal{B}_s, \mathcal{B}_r, \mathsf{txdata}_s, \sigma_s, \mathsf{txdata}_r, \sigma_r)$
    assert $\mathsf{sVer}(\mathcal{B}_s, \mathsf{txdata}_s, \sigma_s) \wedge \mathsf{sVer}(\mathcal{B}_r, \mathsf{txdata}_r, \sigma_r)$
            $\wedge\ \mathsf{VerTxn}(\mathcal{B}_s, \mathcal{B}_r, \mathsf{txn}, \mathsf{Ledger}[\mathcal{B}_s, \mathcal{B}_r])$
    TXID $\leftarrow$ TXID $\cup$ {txid}
    Ledger $\leftarrow$ Ledger || (txid, txn)
    broadcast ("postTxn", txid, txn) to all banks

On receive ("abortTxn", abort) from $\mathcal{B}$:
    Parse abort $\to$ (txid, $(C, e, \mathit{proof}), \mathit{pf}^\star$)
    assert txid $\notin$ TXID
    assert $\mathsf{Ver}(\mathit{ePK}, \mathsf{Ledger}[\mathcal{B}], C, e, \mathit{proof})$
    assert $\mathit{pf}^\star$ proves $e$ is a no-op
    TXID $\leftarrow$ TXID $\cup$ {txid}
    Ledger $\leftarrow$ Ledger || (abort)
    broadcast ("abortTxn", abort) to all banks
```

Figure 2.5: Solidus ideal ledger with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$. $\mathsf{Ledger}[\mathcal{B}_s, \mathcal{B}_r]$ denotes the most recent PVORM states for each bank in Ledger, and VerTxn verifies all proofs associated with a given transaction, which requires the public keys and preceding PVORM state of each bank involved.

path is accessed, $\mathcal{U}_s$ likely is.

In order to prevent this information leakage, $\mathcal{B}_s$ must post some PVORM update to the ledger after sending the update to $\mathcal{B}_r$ before initiating any other transaction. If the original transaction settles that includes exactly such an update. Otherwise $\mathcal{B}_s$ can invoke "abortTxn" with a dummy update on the same tree path, thus invalidating any information $\mathcal{B}_r$ may have gained.

With these ideal functionalities defined, we can now present the main Solidus protocol, **Prot**$_{\text{Sol}}$, in Figure 2.5. We note that the environment $\mathcal{Z}$ is a standard UC framework entity that represents all behavior external to the protocol execution. ($\mathcal{Z}$ feeds input to and collects outputs from protocol parties and the adversary.)

To execute a transaction in **Prot**$_{\text{Sol}}$, a user executes "beginTxn", which sends a "requestTxn" request to the user's bank. The bank verifies the request, updates

```
$\mathbf{Prot}_{\text{Sol}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}\right]$

User $\mathcal{U}_i$:
On input ("beginTxn", $\mathcal{U}_j$, $\$v$) from environment $\mathcal{Z}$:
    Let $\mathcal{B}_i = \mathsf{bank}(\mathcal{U}_i)$ and $\mathcal{B}_j = \mathsf{bank}(\mathcal{U}_j)$
    Generate random unique txid
    Encrypt $c_r = \mathsf{Enc}(ePK_j, pk_j)$ and $c_v = \mathsf{Enc}(ePK_i, \$v)$
    $\sigma = \mathsf{hSign}(sk_i, ePK_j, (\mathsf{txid}, c_r, c_v))$
    send ("requestTxn", txid, $ePK_j, c_r, c_v, \sigma$) to $\mathcal{B}_i$

Bank $\mathcal{B}_i$:
On receive ("initBank", $ePK_j, sPK_j, C_j$) from $\mathcal{F}_{\text{Init}}$:
    assert $B[\mathcal{B}_j]$ is not set
    $B[\mathcal{B}_j] \leftarrow (ePK_j, sPK_j, C_j)$

On receive ("postTxn", txid, txn) from $\mathcal{F}_{\text{Ledger}}$:
    Retrieve $(\mathcal{B}_s, C'_s)$ and $(\mathcal{B}_r, C'_r)$ from txn
    if $(\mathcal{B}_i = \mathcal{B}_s$ or $\mathcal{B}_i = \mathcal{B}_r)$, then $Pend \leftarrow \bot$
    Update $B[\mathcal{B}_s] \leftarrow (ePK_s, sPK_s, C'_s)$
           $B[\mathcal{B}_r] \leftarrow (ePK_r, sPK_r, C'_r)$

On input ("abortPend") from environment $\mathcal{Z}$:
    assert $Pend \neq \bot$
    Retrieve $\mathcal{U}_i$ at $\mathcal{B}_i$ from $Pend$
    Update $(C'_i, e, proof) \leftarrow \mathsf{Update}(eSK_i, (\mathcal{U}_i, 0), C_i)$
    Generate ZK-proof $pf^\star$ that $e$ encrypts 0-value change.
    send ("abortTxn", (txid, $\mathcal{B}_i, (C'_i, e, proof), pf^\star$) to $\mathcal{F}_{\text{Ledger}}$

On receive ("abortTxn", abort) from $\mathcal{F}_{\text{Ledger}}$:
    Parse (txid, $\mathcal{B}_j, (C'_j, e, proof), pf^\star) \leftarrow$ abort
    if $\mathcal{B}_j = \mathcal{B}_i$
        assert $Pend = (\mathsf{txid}, \_)$
        $Pend \leftarrow \bot$
    $B[\mathcal{B}_j] \leftarrow (ePK_j, sPK_j, C'_j)$
    if $Pend \neq \bot$ and $\mathcal{B}_j$ is the other bank in $Pend$
        Execute "abortPend" as described above
```

```
On receive ("requestTxn", txid, $ePK_s, c_v, c_r, \sigma$) from $\mathcal{U}_s$:
    assert ($Pend = \perp$) $\wedge$ (txid is unique)
            $\wedge$ hVer($ePK_i$, (txid, $c_v, c_r$), $\sigma$)
            $\wedge$ ((($\alpha, \beta$), _) $\leftarrow \sigma$ : Dec($eSK_i$, ($\alpha, \beta$)) $= pk_s$)
    Decrypt $\$v$ = Dec($eSK_i, c_v$)
    assert $0 \leq \$v \leq M_i[\mathcal{U}_s]$
    Update ($C_i', e_s, proof_s$) $\leftarrow$ Update($eSK_i$, ($\mathcal{U}_s, -\$v$), $C_i$)
    Let $c_v'$ = Enc($ePK_j, \$v$)
    Generate $txdata_s$ containing:
      • (txid, ($c_v, c_r$), $\sigma, c_v'$)
      • ($C_i', e_s, proof_s$)
      • RangePf($e_v, t$)
      • Proof that $e_s$ updates $\mathcal{U}_s$ by amount in $c_v$
      • Proof that $c_v$ and $c_v'$ encrypt the same value
    $Pend \leftarrow txdata_s$
    $\sigma_s$ = Sign($sSK_i, txdata_s$)
    send ("aprvSendTxn", txid, $txdata_s, \sigma_s$) to $\mathcal{B}_j$

On receive ("aprvSendTxn", txid, $txdata_s, \sigma_s$) from $\mathcal{B}_j$:
    assert ($Pend = \perp$) $\wedge$ (txid is unique)
            $\wedge$ sVer($sPK_j, txdata_s, \sigma_s$)
            $\wedge$ all proofs in $txdata_s$ are valid
    Retrieve (txid, ($c_v, c_r$), $\sigma, c_v'$) from $txdata_s$
    Decrypt $\$v \leftarrow$ Dec($eSK_i, c_v'$)
    assert txid is unique and $\$v \geq 0$
    Decrypt $pk_r$ = Dec($eSK_i, c_r$)
    Update ($C_i', e_r, proof_r$) $\leftarrow$ Update($eSK_i$, ($\mathcal{U}_r, \$v$), $C_i$)
    Generate $txdata_r$ containing:
      • (txid, ($c_v, c_r$), $\sigma, c_v'$)
      • ($C_i', e_r, proof_r$)
      • Proof that $e_r$ updates account $c_r$ by value $c_v'$
    $Pend \leftarrow$ (txid, $txdata_r$)
    $\sigma_r \leftarrow$ Sign($sSK_i, txdata_r$)
    Let txn = ($\mathcal{B}_j, \mathcal{B}_i, txdata_s, \sigma_s, txdata_r, \sigma_r$)
    send ("aprvRecvTxn", txid, txn) to $\mathcal{F}_{\text{Ledger}}$
```

Figure 2.5: $\mathcal{F}_{\text{Ledger}}$-hybrid protocol for Solidus with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$. For simplicity we omit operations to open and close accounts.

its PVORM, signs the update, and forwards it to the recipient's bank. That bank similarly verifies, updates, and signs before posting the completed transaction to $\mathcal{F}_{\text{Ledger}}$. For simplicity the sending bank performs all updates and sends them to the receiving bank. In practice both banks can update their respective PVORMs in parallel as implied by Figure 2.3.

The protocol also contains operations for two other purposes: handling transaction aborts described above and updating other banks' states when they post updates to $\mathcal{F}_{\text{Ledger}}$.

2.4.2 Security Definition

To demonstrate the security of $\textbf{Prot}_{\text{Sol}}$, we need a notion of how a secure Solidus protocol operates. We define this as an ideal functionality $\mathcal{F}_{\text{Sol}}$ presented in Figure 2.6. For an adversary $\mathcal{A}$ and environment $\mathcal{Z}$, we let $\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)$ denote the transcript of $\mathcal{A}$ when interacting with $\textbf{Prot}_{\text{Sol}}$. We let $\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)$ be the transcript produced by a simulator $\mathcal{S}$ when run in the ideal world with $\mathcal{F}_{\text{Sol}}$. This allows us to define security as follows.

Definition 2.1. We say that Solidus *securely emulates* $\mathcal{F}_{Sol}$ if for all real-world PPT adversaries $\mathcal{A}$ and all environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$,

$$\Big|\Pr\left[\mathcal{D}\left(\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)\right) = 1\right] \\ - \Pr\left[\mathcal{D}\left(\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)\right) = 1\right]\Big| \leq negl(\lambda).$$

This definition leads to the following theorem, which we prove in Section 2.10.

Theorem 2.1. *The Solidus protocol* $\textbf{Prot}_{Sol}$ *satisfies Definition 2.1 assuming a DDH-hard group in the ROM.*

```
$\mathcal{F}_{\text{Sol}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}, \{\mathcal{U}_i^{\$}\}_{i=1}^{\ell}\right]$

Init
  Initialize $T$ to empty
  Initialize $V[\mathcal{U}_i] \leftarrow 0$ for $i \in [1, n]$

On receive (“requestTxn”, $\mathcal{U}_r$, $\$v$) from $\mathcal{U}_s$:
  assert $\$v \geq 0$
  Generate unique txid
  $T$[txid] $\leftarrow$ ($\mathcal{U}_s, \mathcal{U}_r, \$v$, “req”)
  send txid to $\mathcal{U}_s$
  send (“req”, txid, $\mathcal{U}_s$, bank($\mathcal{U}_r$), $\$v$) to bank($\mathcal{U}_s$)

On receive (“aprvSendTxn”, txid) from $\mathcal{B}_s$:
  Retrieve ($\mathcal{U}_s, \mathcal{U}_r, \$v, f$) $\leftarrow$ $T$[txid]
  assert $f$ = “req” and $\mathcal{B}_s$ = bank($\mathcal{U}_s$)
  $T$[txid] $\leftarrow$ ($\mathcal{U}_s, \mathcal{U}_r, \$v$, “aprv”)
  send (“aprv”, txid, $\mathcal{B}_s, \mathcal{U}_r, \$v$) to bank($\mathcal{U}_r$)

On receive (“aprvRecvTxn”, txid) from $\mathcal{B}_r$:
  Retrieve ($\mathcal{U}_s, \mathcal{U}_r, \$v, f$) $\leftarrow$ $T$[txid]
  assert $f$ = “aprv” and $\mathcal{B}_r$ = bank($\mathcal{U}_r$)
  Remove $T$[txid] mapping
  Retrieve $\$b_s \leftarrow V[\mathcal{U}_s]$, $\$b_r \leftarrow V[\mathcal{U}_r]$
  assert $\$b_s \geq \$v$ or $\mathcal{U}_s = \mathcal{U}_i^{\$}$ for some $i$
  $V[\mathcal{U}_s] \leftarrow \$b_s - \$v$
  $V[\mathcal{U}_r] \leftarrow \$b_r + \$v$
  // Reveal identities of asset notaries and banks
  Let $\mathcal{P}_s = \mathcal{U}_s$ if $\mathcal{U}_s = \mathcal{U}_i^{\$}$, bank($\mathcal{U}_s$) otherwise
  Let $\mathcal{P}_r = \mathcal{U}_r$ if $\mathcal{U}_r = \mathcal{U}_j^{\$}$, bank($\mathcal{U}_r$) otherwise
  broadcast (“postTxn”, txid, $\mathcal{P}_s \rightarrow \mathcal{P}_r$) to all banks

On receive (“abortTxn”, txid) from $\mathcal{B}$:
  if txid has been seen before // Can “abort” nonexistent transactions
    Retrieve ($\mathcal{U}_s, \mathcal{U}_r$, _, _) $\leftarrow$ $T$[txid]
    assert $\mathcal{B}$ = bank($\mathcal{U}_s$) or $\mathcal{B}$ = bank($\mathcal{U}_r$)
    Remove $T$[txid] mapping
  broadcast (“abortTxn”, txid, $\mathcal{B}$) to all banks
```

Figure 2.6: Ideal functionality for the Solidus system with banks $\{\mathcal{B}_i\}$, users $\{\mathcal{U}_i\}$, and asset notaries $\{\mathcal{U}_i^{\$}\}$. For simplicity we assume a fixed set of accounts for each bank.

In order to prove Theorem 2.1 in the Universal Composability (UC) framework [34], we assume Solidus employs only universally composable (UC) NIZKs. Prior work [13] demonstrates that GSPs can be transformed into UC-NIZKs by using the Fiat-Shamir heuristic and including a ciphertext of the witness under a public key provided by a common initializer. As Solidus already employs this trusted initialization and includes ciphertexts of most operations anyway, the performance impact of ensuring UC-NIZKs is minimal.

2.5 Optimizations

In addition parallelizing operation, there are several optimizations which make Solidus more practical. Some of these optimizations are only appropriate for certain use cases, but they may result in significant speedups when applicable. We include the simpler optimizations in our evaluation in Section 2.6.

2.5.1 Precomputing Randomization Factors

A large computational expense in Solidus is re-randomizing ciphertexts while updating a PVORM. Fortunately, El Gamal allows us to re-randomize ciphertexts by combining them with fresh encryptions of the group identity. That is, in a group $G = \langle g \rangle$ with key pair $(pk = g^{sk}, sk)$ and a ciphertext $c = (\alpha, \beta)$, we can re-randomize c by picking a random $r \leftarrow \mathbb{Z}_{|G|}$ and computing $c' = (\alpha \cdot pk^r, \beta \cdot g^r)$.

Computing (pk^r, g^r) only requires knowledge of the group G, the generator g, and a bank's public key pk, none of which change. This means we can precompute these unit ciphertexts and re-randomize by multiplying in a precomputed value.

Since the system can continue indefinitely, it must continue generating these randomization factors. Many financial systems have predictable high and low load times (e.g., very light traffic at night), so they can utilize otherwise-idle hardware for this purpose during low-traffic times. If the precomputation generates more randomization pairs than the application consumes over a modest time frame (e.g. a day), we can drastically improve performance.

2.5.2 Reducing Verification Overhead

As we see in Section 2.6, proof verification is quite expensive. In the basic protocol, the ledger consensus nodes must each verify every transaction. As more banks join the system this increases the load on the consensus nodes—which may be the banks. By strengthening trust assumptions slightly, we can omit much of this online verification and increase performance. We present two strategies that rely on different assumptions.

Threshold Verification. In the financial industry, there is often a group of entities (e.g., large banks and regulators) who are generally trusted. If a threshold number of these entities verify a transaction, this could give all other consensus nodes—often other banks—confidence that the transaction is valid, allowing them to accept it without further verification. Once the threshold is reached, each other node need only verify the signatures of the trusted entities that verified the transaction, which is far faster than performing a full verification. If the group of trusted entities is significantly larger than the threshold or those entities have much more capacity than others, this strategy will improve system scaling.

Full Offline Verification. In some cases banks can be treated as covert adversaries. That is, they will attempt to learn extra information, but they will subvert the protocol only if attribution is impossible. This situation could arise if each Solidus bank is controlled by a large commercial bank. While a bank may wish to learn as much information as possible, the cost of being caught misbehaving is high enough to deter attributable protocol deviations.

Under these assumptions we can omit online verification entirely. The verifiability of a ledger-based system remains in place, so if a bank submits an invalid transaction or proof, post hoc identification of the faulty transaction and offending bank is trivial. Thus, in this covert adversary model, banks will only submit valid transactions and proofs, meaning that the ledger can accept transactions without first verifying the associated proofs first.

2.5.3 Transaction Pipelining

Solidus requires sequential processing of transactions at a single bank because PVORM updates must be sequential to generate valid proofs. Given transactions T_1 followed by T_2, in order for $\mathcal{B}$ to process T_2 it needs the PVORM state following T_1. It does not, however, need the associated proofs. Therefore, if $\mathcal{B}$ assumes T_1 will settle—because faults are rare—it can start processing T_2 early while generating proofs for T_1. While this technique will not reduce transaction latency, it can drastically increase throughput. Moreover, determining the updated PVORM state requires primarily re-randomizing ciphertexts, making this optimization particularly effective when combined with precomputation (Section 2.5.1).

When failures do occur, it impacts performance but not correctness. If T_1 aborts for any reason, T_2 will not yet have settled since T_1 would have to settle first. This means $\mathcal{B}$ can immediately identify the problem and reprocess T_2—and any following transactions—without T_1. This reprocessing may lead to significant, but temporary, performance degradation, meaning this optimization is only appropriate when failure are rare if each transaction is posted individually to the ledger.

We can alleviate some of this performance penalty by bundling transactions into blocks, as in systems like Bitcoin. If T_1 aborts, instead of reprocessing T_2, $\mathcal{B}$ can include a rollback operation later in the same block. This rollback must provably revert any changes executed by T_1's update, thus allowing verifiers to check that T_1 was never processed at all. There is, however, no need to recompute T_2 as long as the rollback can be placed after it while remaining in the same block as T_1.

2.6 Experiments

We now present performance results for our PVORM and Solidus prototypes. We implemented Solidus in 4300 lines of Java code, 2000 of which form the PVORM. We use BouncyCastle [24] for crypto primitives and Apache ZooKeeper [7] for distributed consensus. We ran all experiments on `c4.8xlarge` Amazon EC2 instances and employed the precomputation optimization (Section 2.5.1). These benchmarks do not include the precomputation time.

We emphasize that our performance results employ an unoptimized implementation and only *one server per bank*, highly limiting our parallelism. Solidus is designed to be *highly parallelized*, allowing it to scale up using multiple servers per bank to achieve *vastly superior performance* in practice.

2.6.1 PVORM Performance

We measured the concrete performance of PVORM Update and Ver operations under different configurations and levels of parallelism.

Bucket and Stash Size. Figure 2.7 shows the single-threaded performance of our PVORM as we vary bucket and stash sizes. As expected, larger buckets are slower and runtime grows linearly with the stash size. As bucket and stash sizes determine the chance of stash overflow, this measures the performance-privacy trade-off.

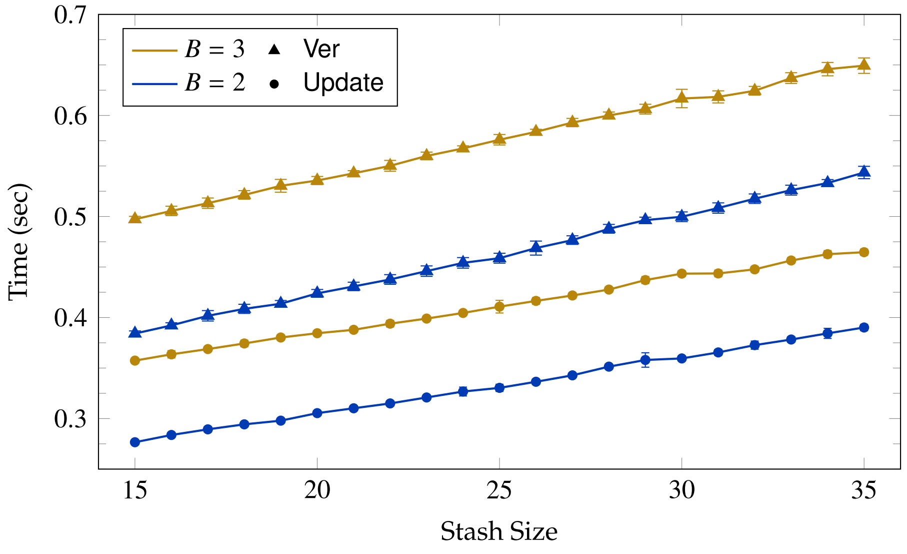


Figure 2.7: PVORM performance with capacity 2^{15} for buckets of size $B = 2$ and $B = 3$ as stash size varies.

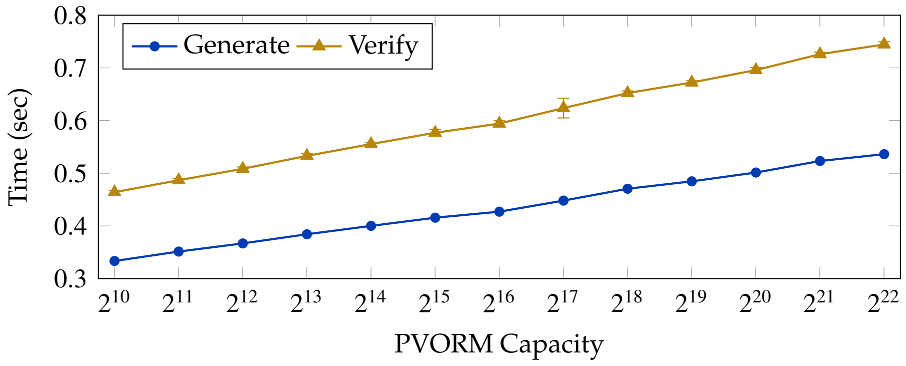


Figure 2.8: PVORM capacity scaling with buckets of size 3 and stash of size 25.

Tree Depth. Figure 2.8 shows the single-threaded performance of our PVORM as the capacity scales. As expected, the binary tree structure results in clearly logarithmic scaling.

Parallelism. Our PVORM construction supports highly parallel operation. One update contains many NIZKs that can be created or verified independently. Figure 2.9 shows the performance for one PVORM with varying numbers of worker threads. In each test, there is one coordination thread, which does very little work.

Because the proof of each pairwise swap can be computed or verified indepen-

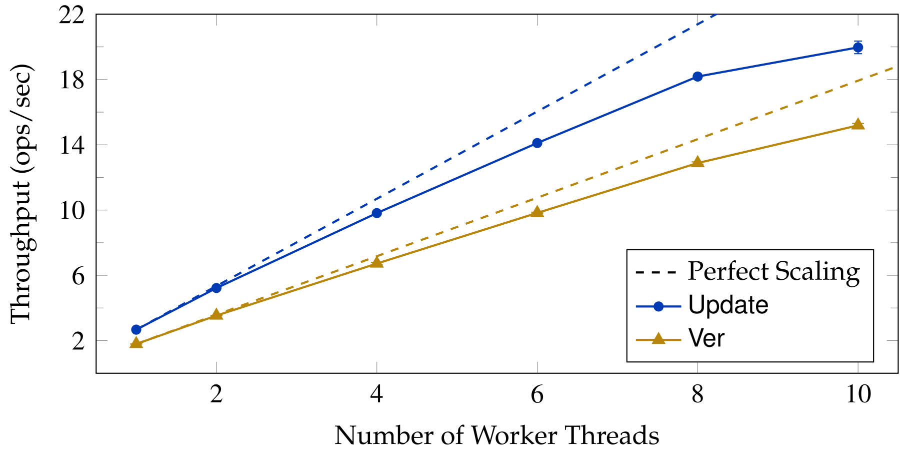


Figure 2.9: Parallel PVORM performance using size 3 buckets, a size 25 stash, and capacity of 2^{15}. Dashed lines show perfect scaling where all computation is parallelized with no overhead.

dently, we expect performance to scale well beyond 10 threads—possibly as high as 100. We stop at 10 for a combination of two reasons. First, PVORM operations are CPU-bound, so adding threads beyond the number of CPU cores produces no meaningful speedup. Second, our prototype implementation does not distribute to multiple hosts and scales poorly to multi-CPU architectures. Since `c4.8xlarge` EC2 instances have two 10-core CPUs, we present scaling to only 10 worker threads. Note that with 10 worker threads there are 11 total threads, so some work may not be effectively parallelized on the same CPU. This likely explains some of the reduced scaling in that case.

Proof Size and Memory Usage. For a PVORM with size 3 buckets, a size 25 stash, and capacity 2^{15}, a single PVORM update with proofs is 190 KB (or 114 KB if compressed[7]). To generate an update, our prototype requires a complete copy of the PVORM in memory. Despite this, memory consumption peaks at only 880 MB.

[7] An elliptic curve point is an ordered pair of elements of $\mathbb{F}_p$. Points can be compressed to a single bit and a field element, but decompression imposes nontrivial overhead.

2.6.2 Solidus System Performance

We present performance tests of a fully distributed Solidus system with 2 to 12 banks. Each bank runs on its own `c4.8xlarge` EC2 instance and maintains a PVORM with size 3 buckets, as size 25 stash, and capacity 2^{15}. These parameters allow realistic testing, with a stash overflow probability of around 2^{-64}. To maintain the ledger, each bank's host also runs a ZooKeeper [7] node. We make no attempt to tune ZooKeeper or optimize off-ledger communication.

To test this configuration we fully load each bank with both incoming and outgoing transactions. As explained in Section 2.5.2, in some settings transaction verification can occur offline, so we also test performance with online verification turned off.

Figure 2.10 contains the results of these tests. With online verification, performance improves until all CPUs are saturated verifying third-party transactions, at which point scaling slows. Using offline verification, transactions settle faster and additional banks impose lower overhead on existing banks, improving scaling.

These results could be further improved by having each bank distribute verification cross multiple machines, improving capacity and throughput. Pipelining transactions (as described in Section 2.5.3) could improve throughput substantially if banks also distributed proof generation across multiple hosts. (Such distribution is unlikely to provide any benefit without pipelining.) Implementing this distribution introduces complex systems engineering challenges that are orthogonal to the technical innovations introduced by Solidus, so we neither implement nor benchmark these options.

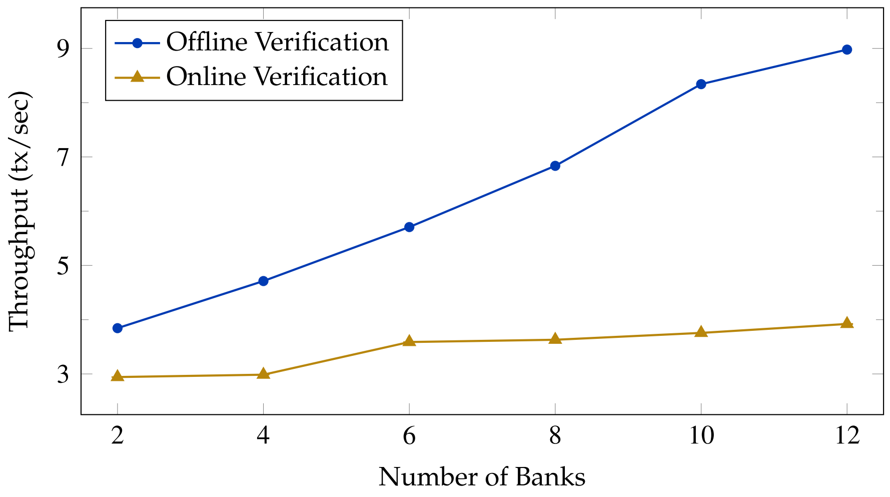


Figure 2.10: Solidus performance distributed using ZooKeeper. Each bank is a ZooKeeper node and maintains a PVORM with size 3 buckets, a size 25 stash, and capacity 2^{15}.

2.6.3 zk-SNARK Comparison

We finally compare our prototype's performance to that of a PVORM implemented with zk-SNARKs. This approach has succinct proofs and short verification times, but costly proof generation.

Taking the Circuit ORAM PVORM construction and converting all proofs to zk-SNARKs would be needlessly expensive. As zk-SNARKs can prove correct application of an arbitrary circuit [14], we use a compact Merkle tree structure. Each account is stored at the leave of a standard Merkle hash tree, the root of which is posted to the ledger. To update the PVORM, a bank updates one account to a valid value and modifies the Merkle tree accordingly. It then produces a zk-SNARK that it properly performed the update and verified the requester's signature. The root of the new Merkle tree is the new PVORM state and the zk-SNARK is the proof.

We implemented this construction using a security level equivalent to our GSP-based PVORM.[8] Table 2.1 shows its performance running on a `c4.8xlarge` EC2

[8]Both hash with SHA-256. The GSP-based PVORM uses El Gamal with the secp256k1 curve and

	Number of Threads		
	1	4	36
Proof Time (sec)	65.45	24.53	13.76
Verification Time	0.0065 sec		
Proof Size	288 bytes		
Peak Memory Use	7.2 GB		

Table 2.1: Performance of PVORM using zk-SNARKs.

instance. While verification is extremely fast, even highly parallel proof generation is more than 200 times slower than the GSP PVORM. For this to improve overall system throughput, the system would need to verify every proof around 200 times. In our expected use-case, at most tens of banks would maintain the ledger, so this is significantly slower. Moreover, additional hardware can allow banks to verify numerous GSP transactions in parallel but provides little benefit to zk-SNARKs.

2.7 Related Work

We now compare Solidus to related work on cryptocurrencies and transaction confidentiality. We omit related work on ORAM, which was covered in Sections 2.1.3 and 2.3.

Anonymous cryptocurrencies. Anonymous e-cash was proposed by Chaum [41, 42] and refined in a long series of works, e.g., [30, 31, 78]. In these schemes, trust is centralized. A single authority issues and redeems coins that are anonymized using blind signatures or credentials. Due to its centralization and technical limitations, such as poor handling of fractional coins and double-spending, e-cash has been largely displaced by decentralized cryptocurrencies.

Zcash, a recently deployed decentralized cryptocurrency, and its academic an-

the SNARK-based PVORM uses RSA-3072. Both provide 128 bits of security.

tecedents [15, 55, 110] and offshoots e.g., Hawk [90], provide strong transaction-graph confidentiality like Solidus. Zcash relies on zk-SNARKs to ensure conservation of money, prevent double spending, and hide both transaction values and the system's transaction graph. Consequently, unlike Solidus, it requires trusted setup, which in practice must be centralized (as multiparty computation for this purpose [17] is impractical). Moreover, as we showed in our exploration of a zk-SNARK variant of Solidus in Section 2.6.3, zk-SNARKs are far more expensive to generate (by two orders of magnitude) than the GSPs used in Solidus. Additionally, Zcash and Hawk do not provide auditability as Solidus does; as designed, they do not record assets on-chain, only commitments.

Alternative schemes such as Monero [112], a relatively popular cryptocurrency, and MimbleWimble [86], a pseudonymous proposal, provide partial transaction-graph concealment. Serious weaknesses in Monero's anonymity have recently been identified, however [115], while MimbleWimble has yet to be deployed or have its confidentiality properties formally analyzed.

Mixes. Mixes partially obscure the transaction graph in an existing cryptocurrency. A number have been proposed and deployed, e.g., [77, 104, 141, 166]. Mixes have a fundamental limitation: they only protect participating users, and thus provide only partial anonymity, resulting in demonstrated weaknesses [114, 158]. As mixes' costs are linear in the number of participants, they do not scale well. In contrast, Solidus achieves strong and rigorously provable transaction-graph confidentiality for all users.

Confidential Transactions. A class of schemes called Confidential Transactions [99, 105, 106] hide transaction amounts, but do not aim at transaction graph privacy. Solidus employs a Confidential Transaction scheme similar to that in [105], but makes more direct use of and inherits the provable security properties of GSPs.

Financial sector blockchain technologies. The financial industry's intense interest in blockchains has led to a number of proposed and deployed systems. These systems support current banking system transaction flows like Solidus. They achieve elements of Solidus, but lack its full set of features. For example, Ripple [40, 136] is a widely deployed scheme for value transfer, but does not aim at the confidentiality properties of Solidus. RSCoin [54], a scheme for central bank cryptocurrency issuance that supports auditability like Solidus, but similarly does not inherently support transaction confidentiality. Other examples are SETLcoin [168], which aims at on-chain trade settlement, like Solidus, but lacks strong transaction-graph confidentiality, and the Digital Asset Platform [61], which provides confidentiality by keeping transaction details off-chain and completely foregoing on-chain settlement and auditability.

2.8 Crypto Primitives

We now describe the basic cryptographic primitives used in Solidus. These primitives operate over a multiplictive cyclic group $G = \langle g \rangle$ of order p determined by (linear in) security parameter λ. As we explain, our building blocks require that the Decisional Diffie-Hellman assumption hold for G. (To prevent sub-group attacks using the Pohlig-Hellman algorithm, p is typically prime.) In our implementation of Solidus, G is the secp256k1 elliptic curve group.

2.8.1 El Gamal Encryption and Account-Balance Representation

The El Gamal cryptosystem (Gen, Enc, Dec) is as follows:

- Gen: $x \overset{\$}{\leftarrow} \mathbb{Z}_q$, $sk \leftarrow x$, $pk \leftarrow g^x$, output (pk, sk)
- Enc(pk, m): if $\neg(m, pk \in G)$, output $\perp$; otherwise $r \overset{\$}{\leftarrow} \mathbb{Z}_q$, $\alpha \leftarrow m \cdot pk^r$, $\beta = g^r$,

output $c = (\alpha, \beta)$

- $\mathsf{Dec}(sk, (\alpha, \beta))$: if $\neg(sk \in \mathbb{Z}_p \wedge \alpha, \beta \in G)$, output $\perp$; output α / β^{sk}

If the Decisional Diffie-Hellman (DDH) problem is hard for G, then El Gamal encryption is semantically secure. El Gamal ciphertexts are malleable, however, a useful feature in our constructions. Specifically, El Gamal has a few useful homomorphisms. Let $(\alpha, \beta) \mapsto m$ mean that (α, β) decrypts to m, i.e., $(\alpha, \beta) = (m \cdot pk^r, g^r)$ for $r \in \mathbb{Z}_p$. Then the following hold:

- *Multiplicative homomorphism:* $(\alpha, \beta) \mapsto m$, $(\alpha', \beta') \mapsto m'$ implies $(\alpha\alpha', \beta\beta') \mapsto mm'$.
- *Additive homomorphism in exponent space:* $(\alpha, \beta) \mapsto g^m$, $(\alpha', \beta') \mapsto g^{m'}$ implies $(\alpha\alpha', \beta\beta') \mapsto g^{m+m'}$.
- *Multiplicative homomorphism in exponent space:* $(\alpha, \beta) \mapsto g^m$ implies $(\alpha^k, \beta^k) \mapsto g^{mk}$.

Observe that re-encryption of a ciphertext $(\alpha, \beta) \mapsto m$ without knowledge of sk is achievable using the multiplicative homomorphism: Let $r \overset{\$}{\leftarrow} \mathbb{Z}_p$, compute a fresh ciphertext $(\alpha', \beta') = (pk^r, g^r) \mapsto 1$, and then let $(A, B) = (\alpha\alpha', \beta\beta')$. Observe that $(A, B) \mapsto (m \times 1) = m$.

Account-Balance Representation. The cryptographic primitives in Solidus rely on a representation of account balances in the exponent space in order to leverage the additive homomorphism in the exponent space illustrated above. Thus an account balance $\$v$ is encoded as $g^{\$v}$ and represented in an El Gamal ciphertext as $(g^{\$v} pk^r, g^r)$ for some $r \in \mathbb{Z}_p$. Decrypting a balance thus requires finding the discrete log of $g^{\$v}$. While in general this is hard in G, if $\$v$ is known to be small (e.g., $0 \leq \$v < 2^{30}$), then the balance can be decrypted using a lookup table of manageable size.

2.8.2 Generalized Schnorr Proofs (GSPs)

Generalized Schnorr Proofs [32] are a type of Σ-protocol, that is, 3-move honest-verifier zero-knowledge (HVZK) proofs (often more specifically defined as special 3-move HVZK proofs with special soundness) [53]. GSP specifically operate over groups for which the discrete log problem and variants are hard. We note that here we consider GSPs only in a cyclic group of prime order, avoiding the caveats of [32] regarding composite-order groups.

Given $x \xleftarrow{\$} \mathbb{Z}_p$ and $y \leftarrow g^x$, there is a simple Σ-protocol to prove knowledge of x to a verifier that knows only $y = g^x$:

- Prover P selects $r \xleftarrow{\$} \mathbb{Z}_p$ and sends $e = g^r$ to Verifier V
- V selects $c \xleftarrow{\$} \mathbb{Z}_p$
- P replies with $s = cx + e$.

Verifier V then checks that $g^s = ey^c$. This protocol is specified in the language of GSPs using notation introduced in [29] as:

$$\mathsf{PoK}\,(x : y = g^x)\,,$$

and is a form of the Schnorr identification protocol.

A more general GSP is possible of the form:

$$\mathsf{PoK}\,(x_1, \ldots, x_k : \mathsf{Pred}(y, (x_1, \ldots, x_k), (y_1, \ldots, y_k)))\,,$$

where Pred is a predicate $y = y_1^{x_1} \cdots y_k^{x_k}$ for a collection of values $y, y_1, \ldots, y_k \in G$ known to the verifier and where the prover proves knowledge of $x_1, \ldots, x_k \in \mathbb{Z}_p$.

It is possible to construct efficient GSPs on conjunctions and disjunctions of such predicates. Additionally, the Fiat-Shamir heuristic [67] can convert GPSs into NIZKs in the Random Oracle Model (ROM) by hashing the prover's message to obtain the challenge. It is also possible to append a supplementary value, which we call a *tag*, to the message to be hashed. The NIZK version of $\mathsf{PoK}(x : y = g^x)$, with tag m, for example, is a Schnorr signature on m. In Solidus, all ZPKs are such NIZKs, a fact we leave implicit in the remainder of the appendix.

2.8.3 Hidden-Public-Key Signatures

In order to authenticate transactions without revealing the sending user, Solidus employs a *hidden-public-key* (HPK) signature scheme. This simple scheme allows a signer to sign with respect to a signing public key pk that is (El Gamal) encrypted under a bank's public key ePK, i.e., a ciphertext $c \xleftarrow{\$} \mathsf{Enc}(ePK, pk)$. An HPK signature scheme $(\mathsf{hGen}, \mathsf{hSign}, \mathsf{hVer})$ with public key ePK is as follows:

- hGen: $sk \xleftarrow{\$} \mathbb{Z}_q$, $pk \leftarrow g^{sk}$, output (pk, sk)
- $\mathsf{hSign}(sk, ePK, m)$: $r \xleftarrow{\$} \mathbb{Z}_p$, $(\alpha, \beta) \leftarrow (pk \cdot ePK^r, g^r)$. Construct a NIZK

  $$pf = \mathsf{PoK}\Big((sk, r) : \Big(g^{sk} \cdot ePK^r = \alpha\Big) \wedge (g^r = \beta)\Big)$$

  with tag m. Output $\sigma = (c = (\alpha, \beta), pf)$.
- $\mathsf{hVer}(ePK, m, \sigma)$: Parse $\sigma = (c, pf)$ and verify pf with ePK, m, c.

An HPK of this form is not terribly useful in and of itself, as the receiver knows only that a valid signature was generated with respect to *some* key, but learns nothing about the key.

The fact that c is an El Gamal ciphertext of pk under ePK, however, makes such signatures useful in two ways. First, when $\mathcal{U}$ requests a transaction, it allows $\mathcal{B}$ to decrypt pk and identify $\mathcal{U}$. Second, it allows $\mathcal{B}$ to generate a plaintext equivalence proof on c and the encryption of the updated account's key. This second property verifies that the user whose balance is updated knows sk, which thus makes this a valid signature.

2.8.4 El Gamal Swaps

The vast majority of the computation required for proof generation and verification in Solidus is devoted to what we call *El Gamal swaps*. The operation **ElGamal-Swap** takes as input an ordered pair of El Gamal ciphertexts $(c_0, c_1) = ((\alpha_0, \beta_0), (\alpha_1, \beta_1))$, a corresponding public key pk, and a value $s \in \{\mathsf{Swap}, \mathsf{NoSwap}\}$. It outputs a fresh ordered pair $((\alpha'_0, \beta'_0), (\alpha'_1, \beta'_1))$, re-encrypted under pk, with the same underlying plaintexts. If $s = \mathsf{NoSwap}$, the plaintext order is the same as the original ciphertexts, otherwise it is swapped. The algorithm is as follows:

```
Algorithm ElGamal-Swap($(c_0, c_1), pk, s$):
  parse $(c_0, c_1) = ((\alpha_0, \beta_0), (\alpha_1, \beta_1))$;
  $r_0 \xleftarrow{\$} \mathbb{Z}_p, r_1 \xleftarrow{\$} \mathbb{Z}_p$;
  if $s = \mathsf{NoSwap}$
    $c'_0 = (\alpha'_0, \beta'_0) \leftarrow (\alpha_0 pk^{r_0}, \beta_0 g^{r_0})$;
    $c'_1 = (\alpha'_1, \beta'_1) \leftarrow (\alpha_1 pk^{r_1}, \beta_1 g^{r_1})$
  else    // $s = \mathsf{Swap}$
    $c'_0 = (\alpha'_0, \beta'_0) \leftarrow (\alpha_1 pk^{r_1}, \beta_1 g^{r_1})$;
    $c'_1 = (\alpha'_1, \beta'_1) \leftarrow (\alpha_0 pk^{r_0}, \beta_1 g^{r_0})$;
  output $(c'_0, c'_1)$
```

It is possible to prove correct execution of **ElGamal-Swap** for an input / output pair (c_0, c_1) and (c'_0, c'_1) via a GSP specified in [85].

In Solidus, due to the fact that an account is represented by a pair of ciphertexts on the public key of an account and the account balance, we in fact need perform *double* El Gamal swaps, meaning that two pairs of ciphertexts are swapped using the same value of s. The proof of correctness involves a straightforward extension of the GSP for a single swap.

A double swap proof requires 13 elliptic curve multiplications, while verification requires 18.

2.8.5 Range Proofs

There are a number of protocols (e.g., [23]) for proving statements of the form $\mathsf{PoK}(x : y = g^x \wedge l_0 \leq x \leq l_p)$.

Drawing on the conceptually simple Confidential Transactions approach [105], we use a GSP to prove that an El Gamal ciphertex $(\alpha, \beta) = (g^{\$v} pk^r, g^r)$ encrypts an account balance $\$v \geq 0$. To prevent modular wraparound, we specifically prove that $\$v \in [0, 2^t)$ for some small integer t. In our prototype, we set $t = 30$.

The GSP we use to accomplish this operates on each bit of $\$v$ separately. For ciphertext (α_i, β_i), to show that $(\alpha_i, \beta_i) \mapsto \$v_i \in \{g^0, g^{2^i}\}$ under public key pk, it suffices to prove:

$$\mathsf{PoK}\left(r_i : \left((\alpha_i / g^{2^i} = pk^{r_i}) \vee (\alpha_i = pk^{r_i})\right) \wedge \beta_i = g^{r_i}\right).$$

Thus the GSP

$$\mathsf{PoK}\left(\{r_i\}_{i=1}^t : \bigwedge_{i=0}^{t-1} \left((\alpha_i / g^{2^i} = pk^{r_i}) \vee ((\alpha_i = pk^{r_i})) \wedge (\beta_i = g^{r_i})\right)\right)$$

proves that $(\alpha, \beta) = \left(\prod_{i=1}^t \alpha_i, \prod_{i=1}^t \beta_i\right)$. Thus if $(\alpha, \beta) \mapsto g^{\$v}$, it must be that $\$v \in [0, 2^t)$ as desired.

This range proof requires $5 + 10t$ elliptic curve multiplications and t encryptions (requiring 2 multiplications each unless precomutation is employed), while verification requires $7 + 12t$ multiplications.

We denote such a proof that ciphertext c encrypts a value in $[0, 2^t)$ (in exponential space) by $\mathsf{RangePf}(c, t)$.

2.8.6 Circuit ORAM

Solidus's primary data structure used to store account balances on the ledger is a PVORM based on the structure of Circuit ORAM [169]. PVORM, however, aims to provide very different guarantees than classical ORAM. An ORAM enables a *client* with limited local memory to maintain a piece of large virtual memory M in a data structure C outsourced to a more powerful external device generically called a *server*. The goal is to enable the client to store M confidentially *with as little local storage as possible*.

An ORAM ensures *access-pattern confidentiality*; despite its ability to observe the client's accesses to C, the server learns nothing (no non-negligible) information about the client's pattern of access to blocks in M. Blocks in C are encrypted using a *symmetric-key* cipher to ensure data confidentiality. Note that encryption alone does not conceal access patterns. M is structured as a set of *blocks* $M[1], M[2], \ldots, M[N]$. Were $C[\mathtt{idx}]$ simply an encryption of the current value of $M[\mathtt{idx}]$, for instance, then the server would know every time the client reads from or writes to $M[\mathtt{idx}]$, as it would see the client access $C[\mathtt{idx}]$.

Thus, to achieve access-pattern confidentiality, ORAM implementations require a more sophisticated approach.

In this approach, C is represented as a tree of depth $L = \log N + 1$ (N is assumed to be a power of 2). Each node in the tree contains a *bucket* that has B slots for storage of blocks, where B is a system parameter. Most of these slots are empty at a given time, an important fact, as we shall see below.

A block takes the form $\mathtt{idx} \| \mathtt{label} \| \mathtt{data}$, where `idx` is the index of a block—the value `idx` corresponding to its virtual memory slot $M[\mathtt{idx}]$, `label` identifies a leaf in the tree along the path to which from the root the block is located in C, and `data` stores the block contents.

The client maintains a small amount of local memory called a *stash*, which is a buffer to handle overflow from C. The client also stores a *position map* PosMap, a data structure such that PosMap[idx] = label. That is, PosMap maps a given block's index idx in M to its corresponding leaf value label. (PosMap can be stored recursively in a separate ORAM on the server to reduce storage overhead, a feature that is not relevant to PVORM.)

Reads and writes involve the same basic operation **Access** by the client on C, which is as follows.

```
Algorithm Access(op):
    // Note: op = ("read", idx) or ("write", idx, data*)
    label ← PosMap[idx];
    {idx||label||data} ← ReadAndRm(idx, label);
    PosMap[idx] $\xleftarrow{\$}$ [0, $N - 1$];
    if op = "read" then data* ← data;
    stash.add({idx||PosMap[idx]||data*});
    Evict();
    output data
```

Here, ReadAndRm reads the full path in C containing the target block and removes the block (re-encrypting blocks along the path), while stash.add performs the obvious operation of adding a block to the stash. Evict can be implemented either randomly or deterministically. The random approach picks two leaves $leaf_l$ and $leaf_r$ uniformly at random from the left and right halves of the three, respectively, and performs what is called an eviction pass in the root-to-leaf paths they define. The deterministic approach (which we adopt in our PVORM construction) does the same, but it selects $leaf_l$ and $leaf_r$ in a rotating deterministic order designed to place eviction passes on consecutive accesses as far away from each other as possible while still rotating through every leaf over enough accesses.

An eviction pass on a given path involves performing swaps on pairs of adjacent path elements one by one from the top to bottom of the tree, with the stash treated as a special "level 0," i.e., sitting above the root. These swaps aim to move blocks down the path to the lowest possible levels. A block is "picked up" and moved through successive swaps to the lowest point such that it remains on the path defined by `label` and there is an empty slot available for it. At this point it is "dropped"—inserted into the bucket at that level. A block may be picked up from the slot into which the last one was dropped or swapping may continue until another block is reached that can be pushed further down the path. The reason for performing evictions on two paths rather than one is to ensure that blocks remain deep enough globally in C to prevent substantial overflow into the stash.

This processing step in Circuit ORAM is in fact quite complicated. The client does not have full local information about where blocks reside in C, and therefore must plan swaps using metadata. (This complication does not arise in PVORM, however, as we explain below.)

Other tree-based ORAMs, such as Path ORAM [160], differ primarily in their use of alternative eviction strategies. The use of swaps in Circuit ORAM is especially conducive to efficient NIZK production in Solidus, however, which is the reason it is used in the Solidus PVORM.

2.9 Solidus PVORM Construction

We now present the details of the PVORM construction used in Solidus and prove that it is a correct, oblivious, and publicly verifiable PVORM. Similar techniques allow construction of a PVORM from any ORAM, ZK proof system, and encryption scheme. Our PVORM is constructed to ensure efficient proof computations in support of high throughputs. For this purpose, we use Circuit ORAM, non-interactive Generalized Schnorr Proofs, and El Gamal encryption.

Recall from above that Circuit ORAM consists of a binary tree of buckets, each containing a fixed number of data blocks. Each location contains an encryption of either a data block or a dummy value. Each logical data block is associated with a single leaf in the tree and physically resides somewhere along the path to that leaf. To access a logical data block, a client reads all blocks along the path to the associated leaf. The client then associates the accessed logical block with a new random leaf, and writes new encryptions of all accessed physical blocks and two other (deterministic) tree paths. During these writes, the client pushes (evicts) existing data blocks as far as possible towards leaves while ensuring that each real data block remains on the path to its associated leaf. These evictions can be done with a number of pairwise swaps of physical memory locations linear in the depth of the tree. We take advantage of the ability to do evictions via pairwise swaps in our PVORM construction.

2.9.1 Construction

In Solidus, each bank maintains its own PVORM to store user account balances. Since the PVORM is uniquely associated with a single bank, we a simple El Gamal key pair for the key pair specified in Section 2.3. Each logical address is specified by an account ID and each data block is itself an account balance. To store these, each data block contains a pair of El Gamal encryptions: one of the account ID and one of the balance. We limit the maximum balance to a relatively small value (e.g., 2^{30} or 2^{40}). This allows us to encrypt balances in exponential space, creating an additive homomorphism, while still permitting decryption (using a lookup table). Let t denote the binary log of the maximum balance.

Thus we interpret M as a map from account IDs to account balances. We define the PVORM update function $f((\mathsf{id}, \$v), M)$ that replaces $M[\mathsf{id}]$ with $M[\mathsf{id}] + \$v$ if id

$(M[\mathsf{id}] + \$v) \in [0, 2^t)$ and is a key in M. Otherwise f is undefined. Intuitively, f updates a single account balance to any value within the valid range.

As noted in Section 2.3, we use a fixed-size public stash instead of the dynamic private one assumed by Circuit ORAM. For simplicity, we merge this stash into the root node of the tree. Each data block in the stash is of the same form as those in the tree. We also employ a distinguished fixed block that exists as a single deterministic block on every path. It may be part of the root/stash or separate.

We now specify the implementation of the operations in Section 2.3. Let (Gen, Enc, Dec) be the standard El Gamal cryptosystem.

Construction 2.1 (Solidus PVORM)**.** We always initialize all balances to 0. The update space U consists of account ID/transaction value pairs, with values being between the max balance and its negative. Initialization proceeds as follows:

```
$\underline{\mathsf{Init}(1^\lambda, \{\mathsf{id}_i\}_{i=1}^n, 0, U):}$
  $(pk, sk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
  for $i \in [1, n]$
    Insert $(\mathsf{id}_i, 0)$ into a Circuit ORAM tree
  Set all unused blocks to $(0, 0)$
  for each block $(\mathsf{id}, 0)$
    Set $C$ at that location to $(\mathsf{Enc}(pk, \mathsf{id}), \mathsf{Enc}(pk, 0))$
    Let $(\alpha, \beta)$ be the encryption of 0
    $pf = \mathsf{PoK}\,(x : (\alpha = \beta^x) \wedge (pk = g^x))$
  return $(pk, sk, C, \{pf\})$
```

If $M = \mathsf{Read}(sk, C)$, then $\mathsf{Update}(sk, u, C)$ is only defined when $f(u, M)$ is defined. This property is easy to check given u, sk, and C, so we omit explicit validation. Let B^F be the distinguished fixed block and assume for simplicity that pk is derivable from sk.

```
Update$(sk, u, C)$:
  $e = (e_{id}, e_v) \xleftarrow{\$} (\mathsf{Enc}(pk, \mathsf{id}), \mathsf{Enc}(pk, \$v))$
  for each block $B_i$ along the path associated with id:
    Let $s$ = Swap if the ID in $B$ is id and NoSwap otherwise.
    $(B^F, B_i') \xleftarrow{\$}$ ElGamal-Swap$((B^F, B_i), pk, s)$
    $pf_i$ = proof of correct swap
  Let $(c_{id}, c_v) \leftarrow B^F$
  $rangePf$ = RangePf$(c_v - e_v, t)$ // (see Section 2.8.5)
  Let $(\alpha, \beta) = (c_{id} - e_{id})$
    $idPf$ = PoK$(x : (\alpha = \beta^x) \wedge (pk = g^x))$
  $B^F \leftarrow (c_{id}, c_v - e_v)$
  for each block $B_i$ along the eviction paths in Circuit ORAM
    Let $s$ = Swap or NoSwap as per Circuit ORAM
    $(B^F, B_i') \xleftarrow{\$}$ ElGamal-Swap$((B^F, B_i), pk, s)$
    $pf_i$ = proof of correct swap
  return $(C', e, (\{B_i'\}, \{pf_i\}, rangePf, idPf))$
```

Verification is performed simply by verifying all NIZKs included in the output of Update and by verifying that the updated B^F was computed correctly between the two sets of swaps.

2.9.2 Security Proofs

We now prove the security of the construction given in the previous section.

Theorem 2.2 (PVORM Correctness)**.** *Construction 2.1 is a* correct *PVORM.*

Proof. The following properties ensure correctness.

- Circuit ORAM is correct when the stash does not overflow and Construction 2.1 modifies Circuit ORAM to leak transaction graph information instead of lose data on overflows.

- El Gamal is correct and includes a multiplicative homomorphism, while we encrypt account balances in exponential space, thus making the homomorphism additive.
- Construction 2.1 employs correct NIZKs and only attempts to prove true statements. □

To prove obliviousness, we provide a hardness reduction to the Decisional Diffie-Hellman (DDH) problem. We do this through a series of reductions. First we consider the following classic definition of CPA security that a cryptosystem $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *CPA secure* if for all PPT adversaries $\mathcal{A}$ there is a negligible function *negl* such that

$$\Big| \Pr\Big[\mathbf{Exp}^{\mathrm{CPA}}(0, \mathcal{A}, \lambda) = 1\Big] - \Pr\Big[\mathbf{Exp}^{\mathrm{CPA}}(1, \mathcal{A}, \lambda) = 1\Big] \Big| \leq negl(\lambda).$$

where $\mathbf{Exp}^{\mathrm{CPA}}(b, \mathcal{A}, \lambda)$ is defined as

```
Experiment $\mathbf{Exp}^{\mathrm{CPA}}(b, \mathcal{A}, \lambda)$:
  $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
  $(m_0, m_1) \xleftarrow{\$} \mathcal{A}(1^\lambda, pk)$
  $c \xleftarrow{\$} \mathsf{Enc}(pk, m_b)$
  return $\mathcal{A}(1^\lambda, c)$
```

It is well known that El Gamal (which Solidus uses) is CPA-secure in a DDH-hard group. We further define *double-CPA security* which we will use to prove obliviousness of our PVORM construction.

Definition 2.2 (Double-CPA Security)**.** A cryptosystem $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *double-CPA secure* if for all PPT adversaries $\mathcal{A}$ there is a negligible *negl* such that

$$\Big| \Pr\Big[\mathbf{Exp}^{\mathrm{2CPA}}(0, \mathcal{A}, \lambda) = 1\Big] - \Pr\Big[\mathbf{Exp}^{\mathrm{2CPA}}(1, \mathcal{A}, \lambda) = 1\Big] \Big| \leq negl(\lambda).$$

where $\mathbf{Exp}^{2\mathrm{CPA}}(0, \mathcal{A}, \lambda)$ is defined as

```
Experiment $\mathbf{Exp}^{2\mathrm{CPA}}(b, \mathcal{A}, \lambda)$:
  $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
  $((m_0, m_0'), (m_1, m_1')) \xleftarrow{\$} \mathcal{A}(1^\lambda, pk)$
  $c \xleftarrow{\$} \mathsf{Enc}(pk, m_b)$
  $c' \xleftarrow{\$} \mathsf{Enc}(pk, m_b')$
  return $\mathcal{A}(1^\lambda, c, c')$
```

We now prove by a hybrid argument that any public-key cryptosystem that is CPA secure (e.g., El Gamal) is double-CPA secure.

Lemma 2.1 (Double-CPA Security). *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a CPA secure public-key cryptosystem. Then it is also a* double-CPA secure *cryptosystem.*

Proof. Assume for contradiction that there is some $\mathcal{A}$ and non-negligible $\nu(\lambda)$ where

$$\Big| \Pr\Big[\mathbf{Exp}^{2\mathrm{CPA}}(0, \mathcal{A}, \lambda) = 1\Big] \\ - \Pr\Big[\mathbf{Exp}^{2\mathrm{CPA}}(1, \mathcal{A}, \lambda) = 1\Big] \Big| \geq \nu(\lambda).$$

We now consider a set of three hybrid experiments. Let $H_0 = \mathbf{Exp}^{2\mathrm{CPA}}(0, \mathcal{A}, \lambda)$, $H_2 = \mathbf{Exp}^{2\mathrm{CPA}}(1, \mathcal{A}, \lambda)$, and

```
Experiment $H_1$:
  $(sk, pk) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
  $((m_0, m_0'), (m_1, m_1')) \xleftarrow{\$} \mathcal{A}(1^\lambda, pk)$
  $c \xleftarrow{\$} \mathsf{Enc}(pk, m_0)$
  $c' \xleftarrow{\$} \mathsf{Enc}(pk, m_1')$
  return $\mathcal{A}(1^\lambda, c, c')$
```

Note that we encrypt m_0 (as in H_0) and m_1' (as in H_2). By the standard hybrid argument $\mathcal{A}$ must have advantage at least $\nu(\lambda)/2$ in distinguishing either between H_0 and H_1 or between H_1 and H_2.

We now construct an adversary $\mathcal{B}$ to break the CPA security of (Gen, Enc, Dec). On input $(1^\lambda, pk)$, $\mathcal{B}$ first runs $\mathcal{A}$ to get (m_0, m_0') and (m_1, m_1'). It then picks a random $i \overset{\$}{\leftarrow} \{0, 1\}$. We handle these cases separately.

- $i = 0$: In this case $\mathcal{B}$ outputs (m_0, m_1). On receipt of challenge c it computes $c' \overset{\$}{\leftarrow} \mathsf{Enc}(pk, m_1')$, submits $(1^\lambda, c, c')$ to $\mathcal{A}$ and returns the result.
- $i = 1$: In this case $\mathcal{B}$ outputs (m_0', m_1'). On receipt of challenge c', it computes $c \overset{\$}{\leftarrow} \mathsf{Enc}(pk, m_0)$ and submits $(1^\lambda, c, c')$ to $\mathcal{A}$ and returns the result.

In the first case, if c encrypts m_0 then this is exactly experiment H_1 and if c encrypts m_1, this is experiment H_2. For the second case, $\mathcal{B}$ has similarly generated either experiment H_0 or H_1. $\mathcal{B}$ will succeed exactly when $\mathcal{A}$ succeeds. Since $\mathcal{A}$ has advantage at least $\nu(\lambda)/2$ in one of these experiments and $\mathcal{B}$ randomly selects which experiment to run, it must be the case that $\mathcal{B}$ succeeds with advantage at least $\nu(\lambda)/4$, which is non-negligible. By assumption, however, (Gen, Enc, Dec) is CPA-secure, so this contradicts our assumption that $\mathcal{A}$ exists. Thus (Gen, Enc, Dec) is double-CPA secure. □

Theorem 2.3 (PVORM Obliviousness)**.** *Construction 2.1 is* oblivious *in the ROM assuming a DDH-hard group.*

Proof. Assume for contradiction that there exists some PPT adversary $\mathcal{A}$ and non-negligible $\nu(\lambda)$ such that

$$\Big| \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(0, \mathcal{A}, \lambda, n, m_0, U) = 1\Big] \\ - \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(1, \mathcal{A}, \lambda, n, m_0, U) = 1\Big]\Big| \geq \nu(\lambda).$$

We now construct an adversary $\mathcal{B}$ that breaks the game $\mathbf{Exp}^{\text{2CPA}}$, as defined in Lemma 2.1, for El Gamal.

First we argue that $\mathcal{A}$ cannot distinguish based solely on observing the pattern of data blocks touched within the Circuit ORAM structure. As noted by Wang, Chan, and Shi [169], each access consists first of accessing a uniformly random path independent from all previous accesses, followed by eviction along two paths chosen independently from the access. Thus $\mathcal{A}$ can only hope to distinguish in this manner by forcing the stash to overflow. Wang, Chan, and Shi additionally note that the probability of stash overflow is negligible in the size of the stash even for a worst-case access pattern. Therefore $\mathcal{A}$ gains at most negligible advantage by observing the Circuit ORAM access structure.

This means that $\mathcal{A}$ must either break the semantic security of El Gamal or the zero-knowledge property of an NIZK. We now assume that $\mathcal{A}$ will make at most $poly(\lambda)$ queries the PVORM oracle for some polynomial $poly$. Using this, we construct a series of hybrid distributions $H_0, \ldots, H_{poly(\lambda)+1}$ modifying how the $\mathbf{Exp}^{\mathrm{Obliv}}$ oracle works.

In hybrid H_0, $\mathcal{O}$ operates exactly as $\mathcal{O}_{1,sk,C}$. In H_1, $\mathcal{O}$ operates the same way except it leverages the fact that we are in the ROM to forge all NIZKs. For H_i with $i \geq 1$, on input (u_0, u_1) from $\mathcal{A}$, the oracle applies update u_1 as in H_1 for the first $i - 1$ queries, after which it applies u_0 instead. Though this may result in invalid updates, the new oracle does not check the validity and applies the update anyway with forged proofs. Because the proofs are forged, it will always succeeded in making this (forged) update. Since, by the definition of the game, $\mathcal{A}$ could not rely on submitting invalid updates in order to distinguish, this cannot improve the advantage at all.

Because we are working in the ROM and all NIZKs are GSPs, $\mathcal{A}$ receives the same view in H_0 and H_1. Whenever the PVORM oracle needs to generate a proof, it first picks a random challenge c and a response. It then computes the commitment com to ensure that the tuple is from the correct distribution, and modifies $\mathcal{A}$'s random oracle so that it receives c when querying that oracle on com. As long as the random oracle has not previously been queried on com, this strategy will work and produce exactly the same distribution as in H_1.

If there is a collision—the random oracle has been queried on com—then the experiment H_1 simply aborts. Fortunately this happens with negligible probability. Specifically, $\mathcal{A}$ makes at most $q(\lambda)$ independent queries to its random oracle for some polynomial q, and $\mathcal{O}$ must forge some constant k number of proofs for each PVORM update. This bounds the probability of collision to $\nu(\lambda) = \frac{k \cdot poly(\lambda)+q(\lambda)}{2^\lambda}$, a negligible function.

We can apply the same argument to $H_{poly(\lambda)+1}$ and the (unnamed) hybrid that corresponds to $\mathcal{O}_{0,sk,C}$ with real proofs. Thus $\mathcal{A}$ can distinguish between H_1 and $H_{poly(\lambda)+1}$ with advantage at least $\nu(\lambda) - 2\nu(\lambda)$. So by a standard hybrid argument, there must be some $i \in [1, poly(\lambda)]$ such that $\mathcal{A}$ can distinguish between H_i and H_{i+1} with advantage at least $\frac{\nu(\lambda)-2\nu(\lambda)}{poly(\lambda)}$. This too is non-negligible. For simplicity, we will denote this advantage $\nu'(\lambda)$.

Next we recall that the secret key is only used to generate NIZKs in Update, meaning an adversary with only the public key can run $\mathcal{A}$ with an oracle that generates any of $H_1, \ldots, H_{poly(\lambda)+1}$. $\mathcal{B}$ is exactly such an adversary.

On input $(1^\lambda, pk)$, $\mathcal{B}$ first guesses a uniformly random $i \in [1, poly(\lambda)]$ and then runs $\mathcal{A}$. $\mathcal{B}$ then handle's $\mathcal{A}$'s PVORM oracle queries as follows. For the first $i-1$ queries (u_0, u_1), $\mathcal{B}$ applies u_1 with forged proofs—as in both H_i and H_{i+1}. Because Update uses sk only for proofs and $\mathcal{B}$ is forging proofs, it can perform the rest of Update properly with only pk. Recall that an update u consists of two plaintexts: an account ID id and a transaction value $\$v$. So to generate its chosen plaintext pairs, $\mathcal{B}$ outputs the updates specified for $\mathcal{A}$'s ith PVORM oracle query. Upon receiving a challenge pair of ciphertexts $e = (c_{\mathsf{id}}, c_v)$, $\mathcal{B}$ performs the rest of Update using that update ciphertext (and forging proofs). For all future PVORM oracle queries after the ith, $\mathcal{B}$ uses update request u_0—as in both H_i and H_{i+1}. When $\mathcal{A}$ terminates with an output, $\mathcal{B}$ outputs the same value.

We now claim that $\mathcal{B}$ has non-negligible advantage in the $\mathbf{Exp}^{\text{2CPA}}$ experiment defined above. With probability at least $\frac{1}{poly(\lambda)}$, $\mathcal{B}$ will pick some i where $\mathcal{A}$ has non-negligible advantage $\nu'(\lambda)$ distinguishing between H_i and H_{i+1}. If $\mathcal{B}$ receives a challenge encryption of u_1, then $\mathcal{A}$ is playing exactly the game in H_i. Similarly, if $\mathcal{B}$ is challenged with an encryption of u_0, then $\mathcal{A}$ sees exactly distribution H_{i+1}. In either case $\mathcal{B}$ will output the correct value exactly when $\mathcal{A}$ does. This means that $\mathcal{B}$ must succeed with advantage at least $\frac{\nu'(\lambda)}{poly(\lambda)}$, which is non-negligible.

By assumption we are working with a DDH-hard group and using El Gamal as our cryptosystem. Thus our cryptosystem is CPA secure, so by Lemma 2.1 no such $\mathcal{B}$ exists. This contradicts our assumption that $\mathcal{A}$ exists and therefore Construction 2.1 must be an oblivious PVORM. $\square$

Theorem 2.4 (PVORM Public Verifiability)**.** *Construction 2.1 is* publicly verifiable *in the ROM.*

Proof. This result follows directly from the fact that our Update specification includes a proof of every operation as well as a range proof. By definition Ver simply verifies all NIZKs produced by Update. Therefore, if an adversary were able to fool Ver, it must be able to forge (at least) one of the proofs produced by Update.

Assume for contradiction that there exists some PPT adversary adversary $\mathcal{A}$ and non-negligible $\nu(\lambda)$ such that

$$\Pr\left[\mathbf{Exp}^{\text{PubVer}}(\mathcal{A}, \lambda, n)\right] \geq \nu(\lambda).$$

We note that Update produces three types of proofs. Thus we construct three new PPT adversaries $\mathcal{B}_R$, $\mathcal{B}_E$, and $\mathcal{B}_S$ that attempt to forge range proofs, proofs of plaintext equivalence on El Gamal ciphertexts, and proofs of correct El Gamal swaps, respectively. They operate as follows.

- $\mathcal{B}_R$: On input (pk, sk), $\mathcal{B}_R$ runs $\mathcal{A}$ and outputs the resulting range proof with associated ciphertexts.
- $\mathcal{B}_E$: On input (pk, sk), $\mathcal{B}_E$ runs $\mathcal{A}$ and outputs the resulting plaintext equivalence proof and associated ciphertexts.
- $\mathcal{B}_S$: On input (pk, sk), $\mathcal{B}_S$ runs $\mathcal{A}$, picks a uniformly random El Gamal swap proof from the output, and outputs that proof and the associated ciphertexts.

Whenever $\mathcal{A}$ forges the one range proof or the one plaintext equivalence proof, $\mathcal{B}_R$ or $\mathcal{B}_E$ succeed, respectively. For $\mathcal{B}_S$, the number of El Gamal swaps executed by Update is fixed for a given PVORM configuration (tree depth, bucket size, and stash size), so if $\mathcal{A}$ forges any El Gamal swap correctness proof, $\mathcal{B}_S$ will succeed with constant probability.

By inspection of the specification of Update and a standard hybrid argument, $\mathcal{A}$ must succeed in forging at least one type of proof with non-negligible probability, hence one $\mathcal{B}_R$, $\mathcal{B}_E$, and $\mathcal{B}_S$ must succeed with non-negligible probability. As we describe in Section 2.8, prior work shows that each of the associated proofs have negligible soundness error in the ROM. Thus no such adversary $\mathcal{A}$ can exist so the Solidus PVORM construction is publicly verifiable in the ROM. $\square$

2.10 Solidus Security Proof

We now provide a proof of Theorem 2.1 that $\mathbf{Prot}_{\text{Sol}}$ is secure.

We assume several simple pieces of behavior not directly specified by the protocol. First, each honest bank will have only one pending transaction at a time. That means that it will not approve a request (as sending or receiving bank) while there is another transaction it has approved that has not yet cleared. In the $\mathcal{F}_{\text{Ledger}}$-hybrid world, this is codified within $\mathbf{Prot}_{\text{Sol}}$, but we simply assume this property in the ideal world. Second, we assume that an honest bank will reply immediately upon receiving a transaction approval request. It may approve or abort the transaction, but it will reply in some fashion. Note that an honest bank may abort a transaction it has already approved in order to maintain availability. Finally, we assume that for an honest bank, whenever an assertion fails, the bank acts exactly as if the message it failed to process was never received.

For simplicity, we omit asset notaries from our proof. Adding them requires only small modification. Initialization must publicly distribute asset notary identities, $\mathcal{F}_{\text{Ledger}}$ must check for valid asset notary signatures, and $\mathbf{Prot}_{\text{Sol}}$ must properly reveal asset notary identities.

Theorem 2.1. *The Solidus protocol* $\mathbf{Prot}_{Sol}$ *satisfies Definition 2.1 assuming a DDH-hard group in the ROM.*

Proof. We prove that $\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)$ and $\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)$ are indistinguishable using a sequence of hybrids. In the following, a probability is *negligible* if it is a negligible function of the security parameter λ.

We define hybrids $H_0, \ldots, H_7$. H_0 is the $\mathcal{F}_{\text{Ledger}}$-hybrid world with $\mathcal{S}$ being a "dummy" simulator that passes all messages through unchanged. H_1 allows $\mathcal{S}$ to simulate $\mathcal{F}_{\text{Ledger}}$. H_2 replaces all proofs generated by honest parties with forgeries and H_3 to replaces the contents of requests and PVORMs with arbitrary values. In H_4 $\mathcal{S}$ simulates the trusted initializer and controls all keys. H_5 isolates $\mathcal{A}$'s set of transaction IDs and H_6 drops any invalid messages from $\mathcal{A}$. Finally H_7 is equivalent to an ideal execution.

Hybrid H_0 contains a dummy simulator that passes messages between $\mathcal{A}$ and honest parties unchanged. This is identical to the $\mathcal{F}_{\text{Ledger}}$-hybrid world.

Hybrid H_1 is the same as H_0 except that $\mathcal{S}$ maintains its own simulated copy of $\mathcal{F}_{\text{Ledger}}$ that behaves as $\mathcal{F}_{\text{Ledger}}$ except for the initialization, which it does not emulate. During initialization, $\mathcal{S}$ passes the actual values sent by $\mathcal{F}_{\text{Ledger}}$ to $\mathcal{A}$ without modification. All other operations are emulated faithfully. We note that all non-initialization operations require only public information (including public keys). When an honest bank posts to $\mathcal{F}_{\text{Ledger}}$, $\mathcal{S}$ copies the message to its own copy, and when $\mathcal{A}$ posts to $\mathcal{F}_{\text{Ledger}}$, $\mathcal{S}$ first simulates the behavior on its copy, and if the post is accepted, it forwards the post to the real $\mathcal{F}_{\text{Ledger}}$.

Since all posts to $\mathcal{F}_{\text{Ledger}}$ are either dropped silently or broadcast in their entirety to all banks, $\mathcal{S}$'s faithful simulation of a copy will result in a view that is identical to real execution.

Hybrid H_2 proceeds as in H_1 except whenever $\mathcal{S}$ receives any proofs or signatures constructed by an honest party—as part of a request, PVORM update, or "postTxn" message from $\mathcal{F}_{\text{Ledger}}$—it stores the real proofs and signatures and replaces them with forgeries. $\mathcal{S}$ sends the forgeries to $\mathcal{A}$ (or the simulated $\mathcal{F}_{\text{Ledger}}$), and if a message containing those proofs would be sent back to an honest party (or forwarded to the real $\mathcal{F}_{\text{Ledger}}$), $\mathcal{S}$ puts the original (real) proofs and signatures back in place.

Note that this forgery and replacement only applies to the specific proofs and signatures constructed by honest parties. Messages from honest parties containing proofs and signatures from $\mathcal{A}$-controlled parties—such as the request signature from an $\mathcal{A}$-controlled user at an honest bank included with the final transaction—have only the honest signatures and proofs replaced. The values computed by $\mathcal{A}$ are left exactly in-tact.

As all proofs in the system are cSE NIZKs, $\mathcal{S}$ can forge proofs that $\mathcal{A}$ will accept and $\mathcal{A}$ still cannot forge proofs with non-negligible probability. Since the only thing that has changed from H_1 is these forged proofs, H_1 and H_2 are computationally indistinguishable.

Hybrid H_3 is much like H_2, but $\mathcal{S}$ also replaces the values of all encryptions generated by honest parties under honest-party keys, including PVORM values. $\mathcal{S}$ replaces these values with randomly-selected values encrypted under the same keys. Again, it saves the real values and real proofs when communicating with honest parties, but it uses the random values with $\mathcal{A}$. Since $\mathcal{S}$ only replaces values that $\mathcal{A}$ did not generate and are encrypted under public keys for which $\mathcal{A}$ does not know the secret key, the semantic security of the encryption scheme guarantees that H_3 is indistinguishable from H_2. The proofs do not present a concern as they were already forged (for the real values) in H_2, so they remain forged (for the random values) in H_3.

Hybrid H_4 differs from H_3 in that $\mathcal{S}$ now emulates the initialization in $\mathcal{F}_{\text{Init}}$. It generates fake keys and PVORMs—from the correct distribution—for all parties and sends those to $\mathcal{A}$ instead of those generated by $\mathcal{F}_{\text{Init}}$. Any encrypted values written by $\mathcal{A}$ will be encrypted under the new (fake) keys for which $\mathcal{S}$ knows the secret key, and any values intended to be read by $\mathcal{A}$ and written by an honest party will be encrypted under a key given to $\mathcal{S}$ by the real $\mathcal{F}_{\text{Init}}$. In either case, $\mathcal{S}$ can decrypt the ciphertext and re-encrypt the plaintext under the other set of keys before passing an honest message to $\mathcal{A}$ or $\mathcal{A}$'s message to an honest party. The same is true for signatures and proofs created by $\mathcal{A}$.

For encryptions under honest-party keys written by honest parties as well as proofs and signatures created by honest parties, $\mathcal{S}$ already replaced those in H_3 with random values and forgeries, respectively, so it simply does the same but under the new (fake) keys.

In this manner, all values, proofs, and signatures viewed by $\mathcal{A}$ in H_4 are the same as those in H_3, but using different encryption/signing keys and different randomness. All encryptions, proofs, and signatures generated by $\mathcal{S}$ to an honest party are similarly the same, but with different randomness. Since the keys and randomness are selected faithfully from exactly the original distributions, H_3 and H_4 are identically distributed.

Hybrid H_5 proceeds as H_4, but $\mathcal{S}$ separates the transaction IDs used by $\mathcal{A}$ from those used by honest parties. Whenever a new request comes from $\mathcal{A}$ with transaction ID $\mathsf{txid}_{\mathcal{A}}$, $\mathcal{S}$ generates a new unique $\mathsf{txid}_{\mathcal{F}}$ to associate with the transaction with honest parties. Whenever a message with a previously-unseen transaction ID $\mathsf{txid}_{\mathcal{F}}$ comes in from an honest party (or $\mathcal{F}_{\text{Ledger}}$), $\mathcal{S}$ generates a new unique $\mathsf{txid}_{\mathcal{A}}$ before forwarding to $\mathcal{A}$ (or the simulated $\mathcal{F}_{\text{Ledger}}$). If, for an incoming message in either direction, $\mathcal{S}$ has seen the ID before, there must be an associated ID in the other set, so it simply uses that.

Since only the transaction IDs have changed and the new IDs are drawn independently from the old IDs using the same methodology, H_4 and H_5 are identically distributed.

Hybrid H_6 is the same as H_5 except $\mathcal{S}$ verifies all proofs and signatures generated by $\mathcal{A}$ on all messages. If any proof or signature fails to verify, $\mathcal{S}$ drops the message and does not forward it. Because all proofs are verified in $\mathbf{Prot}_{\text{Sol}}$ (either by the receiving party or by $\mathcal{F}_{\text{Ledger}}$) before any other processing is done, and $\mathcal{Z}$ dictates that if an assertion fails, the honest party behaves as if the associated message had never arrived, this will not change any message received by $\mathcal{A}$ or the behavior of any honest parties. Similarly, H_6 drops all messages containing transaction IDs which have already been posted to $\mathcal{F}_{\text{Ledger}}$, which honest parties will similarly drop. By the simulation soundness of the NIZKs employed, $\mathcal{A}$ has a negligible probability of forging a proof and thus there is a negligible probability of passing through a message that will be ignored anyway. Hence H_5 and H_6 are computationally indistinguishable.

Hybrid H_7 is the most complex step, as we now replace all honest-party communication with $\mathcal{F}_{\text{Sol}}$. We now describe what $\mathcal{S}$ does in H_7 whenever it would send a message to an honest party in H_6 and whenever it receives a message from $\mathcal{F}_{\text{Sol}}$ in H_7.

- When $\mathcal{S}$ would send a "requestTxn" request to an honest bank $\mathcal{B}$ on behalf of a compromised user $\mathcal{U}_s$ in H_6, $\mathcal{S}$ instead decrypts the values supplied by $\mathcal{A}$ to get the plaintext value $\$v$ and receiving user $\mathcal{U}_r$ and sends ("requestTxn", $\mathcal{U}_r$, $\$v$) to $\mathcal{F}_{\text{Sol}}$ on behalf of $\mathcal{U}_s$. Instead of creating its own $\mathsf{txid}_{\mathcal{F}}$ to link to the $\mathsf{txid}_{\mathcal{A}}$ for this transaction, it uses the one returned by $\mathcal{F}_{\text{Sol}}$.

- When $\mathcal{S}$ would send an "aprvSendTxn" message to and honest bank in H_6, it first checks if there is an associated $\mathsf{txid}_{\mathcal{F}}$ from $\mathcal{F}_{\text{Sol}}$, or if the message is coming unprompted from $\mathcal{A}$. In the first case it sends ("aprvSendTxn", $\mathsf{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\text{Sol}}$. In the second case, it first decrypts the request included with the transaction data, which must be from a compromised user $\mathcal{U}$ at a compromised bank $\mathcal{B}$—otherwise the request would have come through $\mathcal{F}_{\text{Sol}}$ or the proofs would fail to verify and H_6 would already have dropped it. It then submits the associated "requestTxn" message to $\mathcal{F}_{\text{Sol}}$ from $\mathcal{U}$. Upon receiving an associated $\mathsf{txid}_{\mathcal{F}}$ and ("req", $\mathsf{txid}_{\mathcal{F}}$, $\mathcal{U}_s$, $\mathcal{B}_r$, $\$v$), $\mathcal{S}$ sends ("aprvSendTxn", $\mathsf{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\text{Sol}}$.

- When $\mathcal{S}$ would send an "aprvRecvTxn" message to the real $\mathcal{F}_{\text{Ledger}}$ (after passing through the simulated one), it again checks for an associated $\mathsf{txid}_{\mathcal{F}}$ from $\mathcal{F}_{\text{Sol}}$. If none is found, then the transaction must entirely be executed by compromised entities for same reason described above. In this case, $\mathcal{S}$ decrypts the transaction details and executes the entire transaction on $\mathcal{F}_{\text{Sol}}$.

  If an $\mathsf{txid}_{\mathcal{F}}$ is found and $\mathcal{S}$ has seen a "req" response from $\mathcal{F}_{\text{Sol}}$ but not a "aprv" message, then it must be the case that both banks are compromised. As with above, $\mathcal{S}$ finishes the transaction in order, first sending ("aprvSendTxn", $\mathsf{txid}_{\mathcal{F}}$)

and then ("aprvRecvTxn", $\mathsf{txid}_{\mathcal{F}}$).

Finally, if $\mathsf{txid}_{\mathcal{F}}$ is found and $\mathcal{S}$ has seen a "aprv" message from $\mathcal{F}_{\mathsf{Sol}}$ for $\mathsf{txid}_{\mathcal{F}}$, then it simply sends ("aprvRecvTxn", $\mathsf{txid}_{\mathcal{F}}$).

- When $\mathcal{S}$ would send an "abortTxn" message to the real $\mathcal{F}_{\text{Ledger}}$, it again checks if there is an associated $\mathsf{txid}_{\mathcal{F}}$. If there is, it sends ("abortTxn", $\mathsf{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\mathsf{Sol}}$. If not, it generates a random txid and sends ("abortTxn", txid) to $\mathcal{F}_{\mathsf{Sol}}$.

  Note that with negligible probability this new txid will conflict with an existing transaction ID and the abort will not be received, but except with negligible probability this will appropriately create an abort for a non-existent transaction.

- We handle $\mathcal{S}$ receiving ("req", $\mathsf{txid}_{\mathcal{F}}, \mathcal{U}_s, \mathcal{B}_r, \$v$) from $\mathcal{F}_{\mathsf{Sol}}$ in two cases.

  1. If $\mathcal{B}_r$ is honest, then $\mathcal{S}$ acts as it would in H_6 upon receiving a valid ("requestTxn", $\mathsf{txid}_{\mathcal{F}}, ePK_s, c_v, c_r, \sigma$) from $\mathcal{U}_s$, noting that in that case it can decrypt the identity of $\mathcal{U}_s$ and $\$v$, but not the identity of the receiving user.
  2. If $\mathcal{B}_r$ is compromised, while $\mathcal{S}$ would have forwarded a "requestTxn" message in H_6, it does not have sufficient information to create the details of that request correctly. To acquire that information, $\mathcal{S}$ immediately replies to $\mathcal{F}_{\mathsf{Sol}}$ with ("aprvSendTxn", $\mathsf{txid}_{\mathcal{F}}$).

- When $\mathcal{S}$ receives ("aprv", $\mathsf{txid}_{\mathcal{F}}, \mathcal{B}_s, \mathcal{U}_r, \$v$) from $\mathcal{F}_{\mathsf{Sol}}$, we again have three cases.

  1. If $\mathcal{B}_s$ is compromised, then we must have been in case 2 above. Thus $\mathcal{S}$ now has sufficient information to create a complete "requestTxn" message as it would in H_6, so it does so and submits that request to $\mathcal{A}$.
  2. If $\mathcal{B}_s$ is honest but the user who originally requested this transaction $\mathcal{U}_s$ is not, then there must be some $\mathsf{txid}_{\mathcal{A}}$ associated with $\mathsf{txid}_{\mathcal{F}}$ and an associated request. $\mathcal{S}$ can thus manufacture an "aprvSendTxn" message to submit to

$\mathcal{A}$. As in H_6, $\mathcal{S}$ uses the stored request for values created by $\mathcal{U}_s$ and falsifies values created by the honest $\mathcal{B}_s$.

3. If $\mathcal{B}_s$ and the sending user $\mathcal{U}_s$ are both honest, then $\mathcal{S}$ must create a new unique $\mathsf{txid}_{\mathcal{A}}$ and create an "aprvSendTxn" message as in H_6. Note that the values $\mathcal{S}$ could decrypt in H_6 were the identity of $\mathcal{U}_r$ and $\$v$, so it encrypts the correct values for those and falsifies other values.

- When $\mathcal{S}$ receives ("postTxn", $\mathsf{txid}_{\mathcal{F}}$, $\mathcal{P}_s \to \mathcal{P}_r$) from $\mathcal{F}_{\text{Sol}}$, Since this proof does not handle asset notaries, we can assume $\mathcal{P}_s$ and $\mathcal{P}_r$ are both banks. There are three cases to consider.

  First we consider the simplest case: when $\mathcal{P}_r$ is a compromised bank. In this case the transaction will only clear through $\mathcal{F}_{\text{Sol}}$ after $\mathcal{S}$ successfully posts it to (the simulated) $\mathcal{F}_{\text{Ledger}}$. Thus there is nothing to do.

  Next we consider the case where $\mathcal{P}_s$ is a compromised bank but $\mathcal{P}_r$ is honest. Here $\mathsf{txid}_{\mathcal{F}}$ must correspond to $\mathsf{txid}_{\mathcal{A}}$ for the pending transaction in $\mathcal{S}$'s simulation of $\mathcal{P}_r$. In order for the transaction to be approved by the sender in $\mathcal{F}_{\text{Sol}}$, $\mathcal{S}$ must have received and verified ("signRecvTxn", $\mathsf{txid}_{\mathcal{A}}$, txdata_s) from $\mathcal{A}$. At this point $\mathcal{S}$ updates $\mathcal{P}_r$'s simulated PVORM with random values and forged proofs (as in H_6) and posts the full transaction to $\mathcal{F}_{\text{Ledger}}$. We note that $\mathcal{A}$ cannot have already submitted a transaction to $\mathcal{F}_{\text{Ledger}}$ with ID $\mathsf{txid}_{\mathcal{A}}$ since honest banks respond instantly, so this must be in response to approving the sending of a transaction and H_6 would have dropped that message if $\mathsf{txid}_{\mathcal{A}}$ had already been posted to $\mathcal{F}_{\text{Ledger}}$.

  Finally, we consider the case where $\mathcal{P}_s$ and $\mathcal{P}_r$ are both honest. In this case $\mathcal{S}$ manufactures random updates to the respective PVORMs and forges all associated proofs. If $\mathsf{txid}_{\mathcal{F}}$ already corresponds to some $\mathsf{txid}_{\mathcal{A}}$, that means the requesting user was compromised, and $\mathcal{S}$ simply uses that request. Otherwise $\mathcal{S}$ selects a new unique $\mathsf{txid}_{\mathcal{A}}$ and creates a request specification (again with random values and forged proofs). It then posts the result to the simulated $\mathcal{F}_{\text{Ledger}}$. We note that this is precisely the value that would have been posted to the simulated $\mathcal{F}_{\text{Ledger}}$ in H_6.

- When $\mathcal{S}$ receives ("abortTxn", $\mathsf{txid}_{\mathcal{F}}, \mathcal{B}$) from $\mathcal{F}_{\text{Sol}}$, it first checks of $\mathcal{B}$ is compromised. If so, this must be the response after sending an abort to $\mathcal{F}_{\text{Sol}}$ and there is nothing to do. If not, $\mathcal{S}$ checks if there is a known $\mathsf{txid}_{\mathcal{A}}$ already linked to $\mathsf{txid}_{\mathcal{F}}$ and generates a new unique $\mathsf{txid}_{\mathcal{A}}$ otherwise. It then generates an abort operation using random values and forged proofs, as in H_6 and posts it to $\mathcal{F}_{\text{Ledger}}$. It also clears the simulated pending transactions for $\mathcal{B}$ (which will only happen if $\mathsf{txid}_{\mathcal{A}}$ already existed).

Thus we see that each hybrid is computationally indistinguishable from the next, H_0 corresponds to the $\mathcal{F}_{\text{Ledger}}$-hybrid world, and H_7 corresponds to the ideal world. Thus $\textbf{Prot}_{\text{Sol}}$ achieves the desired security. □

2.11 Variants

We now present three variants on the Solidus system based on different architectural primitives. They provide different guarantees and features which we believe are relevant.

2.11.1 zk-SNARK PVORM

Though GSPs are highly efficient to construct, they can be quite large and expensive to verify. In circumstances where the size of proofs or the verification time is more important than generation time, zk-SNARKs provide a good alternative. While we could implement the Circuit ORAM-based PVORM described in Section 2.3 and Section 2.9 using zk-SNARKs, the large numbers of reencryptions would result in very expensive proofs, even if we were to use symmetric-key primitives. Instead, in Section 2.6.3 we describe and evaluated a different construction, based on a Merkle tree, which is much more efficient for zk-SNARKs than use of Circuit ORAM.

Our evaluation in Table 2.1 shows the performance for a single bank update at 128-bit security level, using `libsnark` [16] as the back end for computing the zk-SNARK proofs. The Merkle tree has depth 15 giving the PVORM a capacity of 2^{15} (the same as in our GSP tests). Our implementation includes zk-SNARK-optimized SHA-256 circuits for the Merkle tree, and optimized circuits for RSA-3072 encryption (RSAES-PKCS1-v1_5) and signatures (RSASSA-PKCS1-v1_5 with SHA-256). We used PKCS #1 v1.5 primitives instead of the more up-to-date PKCS #1 v2.2 primitives and alternative public-key schemes for three reasons: they yield less expensive zk-SNARK circuits, they are still used in practice, and they provide a conservative (i.e. competitive) comparison point for GSPs.

When used in Solidus, the zk-SNARK PVORM construction has the clear drawback that the ledger does not contain each user's account balance, even in encrypted form. To compute a user's balance, an auditor would need to parse the transaction ciphertexts, decrypt them and perform all the operations. To reduce such overhead in practice, however, the bank may periodically checkpoint balances. Specifically, it may submit an encrypted version of the Merkle tree leaves, and prove that the encryptions are consistent with a published Merkle tree digest using another zk-SNARK proof. Such a proof is quite expensive to construct, and could only be done periodically, e.g., once per day, without significantly affecting the system throughput. But as transactions are accompanied by ciphertexts, an auditor can start at a checkpoint and then decrypt all subsequent transactions to learn current account balances.

Of course, proof generation times are more important in the applications targeted by Solidus, and in our discussions with blockchain industry technologists, the engineering complexity of zk-SNARKs and trusted setup make them less viable than GSPs today. But zk-SNARKs offer an interesting alternative construction and illustrate what could be a valuable point in the PVORM design space.

2.11.2 Use of Trusted Hardware

Using Intel Software Guard Extensions (SGX) it is possible to construct a much more efficient PVORM. SGX provides a new set of instructions that permits execution of an application inside an *enclave* [6, 80, 107], which protects the application's control-flow integrity and confidentiality against even a hostile operating system. SGX additionally enables generation of *attestations* that prove to a remote party that an enclave is running a particular application (identified as a hash of its build memory).

To reduce the expense of attestations, an enclave can generate a signing key pair and attest to the integrity of the public key [84, 182]. It can then generate the equivalent of a NIZK by simply signing an assertion that it knows a witness to the statement. Trust in SGX then translates to trust in the application and thus its assertions. Verifying an assertion requires only a single digital signature verification.

Using an SGX-based approach, we can build an extremely fast PVORM. We replace the public-key encryption with symmetric-key encryption and all NIZKs with SGX-signed assertions. We can even employ write-only ORAM [21, 137] to further improve performance. Additionally, a PVORM constructed in the *Sealed-Glass Proof* (SGP) model [164] provides security against arbitrarily strong side-channel attacks, provided that the secret signing key remains protected—such as by using a side-channel-resistant crypto library.

While several complications remain to be address (e.g., the need to share keys across enclaves on different hosts in case of failure), we believe that this approach is eminently practical—albeit under the (strong) assumption of trust in Intel and SGX's implementation.

2.11.3 Use of Pedersen Commitments

One of the important features of Solidus is auditability, which is greatly aided by having all account balances encrypted on the ledger. Many financial companies and regulatory agencies are, however, wary to include this information, even in encrypted form [22, 88, 165]. While we believe it would degrade the functionality significantly to omit these encryptions, it is not particularly difficult.

Instead of including encrypted balances on the ledger, banks could instead represent PVORM elements as Pedersen commitments [128]. Unlike El Gamal ciphertexts, Pedersen commitments are perfectly hiding and computationally binding. To implement this, banks would need to retain witnesses for each commitment, which consists of both the account balance and the randomization factor. The bank could then reveal this witness to an auditor in order to prove an account balance, and the proof schemes used with El Gamal ciphertexts would require only slight modification to prove information about the known witnesses.

CHAPTER 3

NONMALLEABLE INFORMATION FLOW CONTROL

An ongoing foundational challenge for computer security is to discover rigorous, yet sufficiently flexible, ways to specify what it means for a computing system to be secure. Such security conditions should be *extensional*, meaning that they are based on the externally observable behavior of the system rather than on unobservable details of its implementation. To allow security enforcement mechanisms to scale to large systems, a security condition should also be *compositional*, so that secure subsystems remain secure when combined into a larger system.

Noninterference, along with many variants [72, 144], has been a popular security condition precisely because it is both extensional and compositional. Noninterference forbids all flows of information from "high" to "low", or more generally, flows of information that violate a lattice policy [59].

Unfortunately, noninterference is also known to be too restrictive for most real systems, which need fine-grained control over when and how information flows. Consequently, most implementations of information flow control introduce *downgrading* mechanisms to allow information to flow contrary to the lattice policy. Downgrading confidentiality is called *declassification*, and downgrading integrity—that is, treating information as more trustworthy than information that has influenced it—is known as *endorsement* [178].

Once downgrading is permitted, noninterference is lost. The natural question is whether downgrading can nevertheless be constrained to guarantee that systems still satisfy some meaningful, extensional, and compositional security conditions. This paper shows how to constrain the use of both declassification and endorsement in a way that ensures such a security condition holds.

Starting with the work of Biba [19], integrity has often been viewed as dual to confidentiality. Over time, that simple duality has eroded. In particular, work on *robust declassification* [12, 43, 120, 177, 178] has shown that in the presence of declassification, confidentiality depends on integrity. It is dangerous to give the adversary the ability to influence declassification, either by affecting the data that is declassified or by affecting the decision to perform declassification. By preventing such influence, robust declassification stops the adversary from *laundering* confidential data through existing declassification operations. Operationally, languages prevent laundering by restricting declassification to high integrity program points. Robust declassification can be enforced using a modular type system and is therefore compositional.

This paper introduces a new security condition, *transparent endorsement*, which is dual to robust declassification: it controls endorsement by using *confidentiality* to limit the possible relaxations of *integrity*. Transparent endorsement prevents an agent from endorsing information that the provider of the information could not have seen. Such endorsement is dangerous because it permits the provider to affect flows from the endorser's own secret information into trusted information. This restriction on endorsement enforces an often-implicit justification for endorsing untrusted inputs in high-integrity, confidential computation (e.g., a password checker): low-integrity inputs chosen by an attacker should be chosen without knowledge of secret information.

A similar connection between the confidentiality and integrity of information arises in cryptographic settings. A *malleable* encryption scheme is one where a ciphertext encrypting one value can be transformed into a ciphertext encrypting a related value. While sometimes malleability is intentional (e.g., *homomorphic* encryption), an attacker's ability to generate ciphertexts makes malleable encryption insufficient to authenticate messages or validate integrity. Nonmalleable encryption schemes [63] prevent such attacks. This paper combines robust declassification and transparent endorsement into a new security condition, *nonmalleable information flow*, which rules out analogous attacks in an information flow control setting.

The contributions of this chapter are as follows:

- We give example programs showing the need for a security condition that controls endorsement of secret information.
- We generalize *robust declassification* to programs including complex data structures with heterogeneously labeled data.
- We identify *transparent endorsement* and *nonmalleable information flow*, new extensional security conditions for programs that mix both declassification and endorsement.
- We present a core language, NMIFC, which provably enforces robust declassification, transparent endorsement, and nonmalleable information flow.
- We present the a formulation of robust declassification as a *4-safety hyperproperty*, and define 4-safety hyperproperties for transparent endorsement and nonmalleable information flow, the first time information security conditions have been characterized as k-safety hyperproperties with $k > 2$.
- We describe our implementation of NMIFC using Flame, a flow-limited authorization library for Haskell and adapt an example of the Servant web application framework, accessible online at `http://memo.flow.limited`.

We organize the paper as follows. Section 3.1 provides examples of vulnerabilities in prior work. Section 3.2 reviews relevant background. Section 3.3 introduces our approach for controlling dangerous endorsements, and Section 3.4 presents a syntax, semantics, and type system for NMIFC. Section 3.5 formalizes our security conditions and Section 3.6 restates them as hyperproperties. Section 3.7 discusses our Haskell implementation, Section 3.8 compares our approach to related work. Sections 3.9 – 3.12 include full details of NMIFC and all proofs.

3.1 Motivating Examples

To motivate the need for an additional security condition and give some intuition about transparent endorsement, we give three short examples. Each example shows code that type-checks under existing information-flow type systems even though it contains insecure information flows, which we are able to characterize in a new way.

These examples employ the notation of the flow-limited authorization model (FLAM) [11], which offers an expressive way to state both information flow restrictions and authorization policies. However, the problems observed in these examples are not specific to FLAM; they arise in all previous information-flow models that support downgrading (e.g., [27, 65, 92, 119, 140, 170, 180]). The approach in this paper can be applied straightforwardly to the decentralized label model (DLM) [119], and with more effort, to DIFC models that are less similar to FLAM. While some previous models lack a notion of integrity, from our perspective they are even worse off, because they effectively allow *unrestricted* endorsement.

In FLAM, principals and information flow labels occupy the same space. Given a principal (or label) p, the notation $p^{\rightarrow}$ denotes the confidentiality projection of p, whereas the notation $p^{\leftarrow}$ denotes its integrity projection. Intuitively, $p^{\rightarrow}$ represents the authority to decide where p's secrets may flow *to*, whereas $p^{\leftarrow}$ represents the authority to decide where information trusted by p may flow *from*. Robust declassification ensures that the label $p^{\rightarrow}$ can be removed via declassification only in code that is trusted by p; that is, with integrity $p^{\leftarrow}$.

Information flow policies provide a means to specify security requirements for

```
1 String$_T$ password;
2
3 bool$_{T^\leftarrow}$ check_password(String$_{T^\rightarrow}$ guess) {
4     bool$_T$ endorsed_guess = endorse(guess, $T$);
5     bool$_T$ result = (endorsed_guess == password);
6     return declassify(result, $T^\leftarrow$);
7 }
```

Figure 3.1: A password checker with malleable information flow

a program, but not an enforcement mechanism. For example, confidentiality policies might be implemented using encryption and integrity policies using digital signatures. Alternatively, hardware security mechanisms such as memory protection might be used to prevent untrusted processes from reading confidential data. The following examples illustrate issues that would arise in many information flow control systems, regardless of the enforcement mechanism.

3.1.1 Fooling a Password Checker

Password checkers are frequently used as an example of necessary and justifiable downgrading. However, incorrect downgrading can allow an attacker who does not know the password to authenticate anyway. Suppose there are two principals, a fully trusted principal T and an untrusted principal U. The following information flows are then secure: $U^\rightarrow \sqsubseteq T^\rightarrow$ and $T^\leftarrow \sqsubseteq U^\leftarrow$. Figure 3.1 shows in pseudo-code how we might erroneously implement a password checker in a security-typed language like Jif [117]. Because this pseudo-code would satisfy the type system, it might appear to be secure.

The argument guess has no integrity because it is supplied by an untrusted, possibly adversarial source. It is necessary to declassify the result of the function (at line 6) because the result indeed leaks a little information about the password. Robust declassification, as enforced in Jif, demands that the untrusted guess be

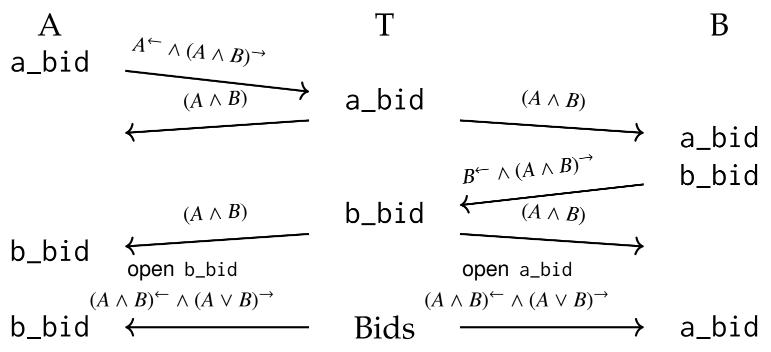


Figure 3.2: Cheating in a sealed-bid auction. Without knowing Alice's bid, Bob can always win by setting `b_bid := a_bid + 1`

endorsed before it can influence information released by declassification.

Unfortunately, the `check_password` policy does not prevent faulty or malicious (but well-typed) code from supplying `password` directly as the argument, thereby allowing an attacker with no knowledge of the correct password to "authenticate." Because `guess` is labeled as secret ($T^{\rightarrow}$), a flow of information from `password` to `guess` looks secure to the type system, so this severe vulnerability could remain undetected. To fix this we would need to make `guess` less secret, but no prior work has defined rules that would require this change. The true insecurity, however, lies on line 4, which erroneously treats sensitive information as if the attacker had constructed it. We can prevent this insecurity by outlawing such endorsements.

3.1.2 Cheating in a Sealed-Bid Auction

Imagine that two principals A and B (Alice and Bob) are engaging in a two-party sealed-bid auction administered by an auctioneer T whom they both trust. Such an auction might be implemented using cryptographic commitments and may even simulate T without need of an actual third party. However, we abstractly specify the information security requirements that such a scheme would aim to satisfy. Consider the following sketch of an auction protocol, illustrated in Figure 3.2:

1. A sends her bid `a_bid` to T with label $A^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$. This label means `a_bid` is trusted only by those who trust A and can be viewed only if *both* A and B agree to release it.

2. T accepts `a_bid` from A and uses his authority to endorse the bid to label $(A \wedge B)^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ (identically, $A \wedge B$). The endorsement prevents any further unilateral modification to the bid by A. T then broadcasts this endorsed `a_bid` to A and B. This broadcast corresponds to an assumption that network messages can be seen by all parties.

3. B constructs `b_bid` with label $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ and sends it to T.

4. T endorses `b_bid` to $A \wedge B$ and broadcasts the result.

5. T now uses its authority to declassify both bids and send them to all parties. Since both bids have high integrity, this declassification is legal according to existing typing rules introduced to enforce (qualified) robust declassification [11, 43, 120].

Unfortunately, this protocol is subject to attacks analogous to *mauling* in malleable cryptographic schemes [63]: B can always win the auction with the minimal winning bid. In Step 3 nothing prevents B from constructing `b_bid` by adding 1 to `a_bid`, yielding a new bid with label $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ (to modify the value, B must lower the value's integrity as A did not authorize the modification).

Again an insecurity stems from erroneously endorsing overly secret information. In step 4, T should not endorse `b_bid` since it could be based on confidential information inaccessible to B—in particular, `a_bid`. The problem can be fixed by giving A's bid the label $A^{\rightarrow} \wedge A^{\leftarrow}$ (identically, just A), but existing information flow systems impose no such requirement.

3.1.3 Laundering Secrets

Wittbold and Johnson [172] present an interesting but insecure program:

```
while (true) do {
    x = 0 [] x = 1;  // generate secret probabilistically
    output x to H;
    input y from H;   // implicit endorsement
    output x ⊕ (y mod 2) to L
}
```

In this code, there are two external agents, H and L. Agent H is intended to have access to secret information, whereas L is not. The code generates a secret by assigning to the variable x a nondeterministic, secret value that is either 0 or 1. The choice of x is assumed not to be affected by the adversary. Its value is used as a one-time pad to conceal the secret low bit of variable y.

Wittbold and Johnson observe that this code permits an adversary to launder one bit of another secret variable z by sending z ⊕ x as the value read into y. The low bit of z is then the output to L.

Let us consider this classic example from the viewpoint of a modern information-flow type system that enforces robust declassification. In order for this code to type-check, it must declassify the value `x ⊕ (y mod 2)`. Since the attack depends on y being affected by adversarial input from H, secret input from H must be low-integrity (that is, its label must be $H^{\rightarrow}$). But if it is low-integrity, this input (or the variable y) must be endorsed to allow the declassification it influences. As in the previous two examples, the endorsement of high-confidentiality information enables exploits.

3.2 Background

We explore nonmalleable information flow in the context of a simplified version of FLAM [11], so we first present some background. FLAM provides a unified model for reasoning about both information flow and authorization. Unlike in previous models, principals and information flow labels in FLAM are drawn from the same set $\mathcal{L}$. The interpretation of a label as a principal is the least powerful principal trusted to enforce that label. The interpretation of a principal as a label is the strongest information security policy that principal is trusted to enforce. We refer to elements of $\mathcal{L}$ as principals or labels depending on whether we are talking about authorization or information flow.

Labels (and principals) have both confidentiality and integrity aspects. A label (or principal) ℓ can be projected to capture just its confidentiality ($\ell^{\rightarrow}$) and integrity ($\ell^{\leftarrow}$) aspects.

The information flow ordering $\sqsubseteq$ on labels (and principals) describes information flows that are secure, in the direction of increasing confidentiality and decreasing integrity. The orthogonal trust ordering $\Rightarrow$ on principals (and labels) corresponds to increasing trustedness and privilege: toward increasing confidentiality and *increasing* integrity. We read $\ell \sqsubseteq \ell'$ as "ℓ flows to ℓ'", meaning ℓ' specifies a policy at least as restrictive as ℓ does. We read $p \Rightarrow q$ as "p acts for q", meaning that q delegates to p.

The information flow and the trust orderings each define a lattice over $\mathcal{L}$, and these lattices lie intuitively at right angles to one another. The least trusted and least powerful principal is $\perp$, (that is, $p \Rightarrow \perp$ for all principals p), and the most trusted and powerful principal is $\top$ (where $\top \Rightarrow p$ for all p). We also assume there is a set of *atomic principals* like `alice` and `bob` that define their own delegations.

Since the trust ordering defines a lattice, it has meet and join operations. Principal $p \wedge q$ is the least powerful principal that can act for both p and q; conversely, $p \vee q$ can act for all principals that both p and q can act for. The least element in the information flow ordering is $\top^{\leftarrow}$, representing maximal integrity and minimal confidentiality, whereas the greatest element is $\top^{\rightarrow}$, representing minimal integrity and maximal confidentiality. The join and meet operators in the information flow lattice are the usual $\sqcup$ and $\sqcap$, respectively.

Any principal (label) can be expressed in a normal form $p^{\rightarrow} \wedge q^{\leftarrow}$ where p and q are CNF formulas over atomic principals [11]. This normal form allows us to decompose decisions about lattice ordering (in either lattice) into separate questions regarding the integrity component (p) and the confidentiality component (q). Lattice operations can be similarly decomposed.

FLAM also introduces the concept of the *voice* of a label (principal) ℓ, written $\nabla(\ell)$. Formally, for a normal-form label $\ell = p^{\rightarrow} \wedge q^{\leftarrow}$, we define voice as follows: $\nabla(p^{\rightarrow} \wedge q^{\leftarrow}) \triangleq p^{\leftarrow}$.[1] A label's voice represents the minimum integrity needed to securely declassify data constrained by that label, a restriction designed to enforce robust declassification.

The Flow-Limited Authorization Calculus (FLAC) [9] previously embedded a simplified version of the FLAM proof system into a core language for enforcing secure authorization and information flow. FLAC is an extension of the Dependency Core Calculus (DCC) [1, 3] whose types contain FLAM labels. A computation is additionally associated with a program-counter label pc which tracks the influences on the control flow and values that are not explicitly labeled.

In this paper we take a similar approach: NMIFC enforces security policies by performing computation in a monadic context. As in FLAC, NMIFC includes a pc

[1]FLAM defines $\nabla(p^{\rightarrow} \wedge q^{\leftarrow}) = p^{\leftarrow} \wedge q^{\leftarrow}$, but our simplified definition is sufficient for NMIFC. For clarity, the operator ∇ is always applied to a projected principal.

label. For an ordinary value v, the monadic term $(\eta_\ell\ v)$ signifies that value with the information flow label ℓ. If value v has type τ, the term $(\eta_\ell\ v)$ has type ℓ says τ, capturing the confidentiality and integrity of the information.

Unlike FLAC, NMIFC has no special support for dynamic delegation of authority. Principals may statically deligate their authority to other principals, adding extra relationships in $\mathcal{L}$. NMIFC then includes traditional declassification and endorsement operations, decl and $\mathsf{endorse}$. We leave to future work the integration of nonmalleable information flow with secure dynamic delegation.

3.3 Enforcing Nonmalleability

Multiple prior security-typed languages—both functional [9] and imperative [12, 43, 120]—aim to allow some form of secure downgrading. These languages place no restriction whatsoever on the confidentiality of endorsed data or the context in which an endorsement occurs. Because of this permissiveness, all three insecure examples from Section 3.1 type-check in these languages.

3.3.1 Robust Declassification

Robust declassification prevents adversaries from using declassifications in the program to release information that was not intended to be released. The adversary is assumed to be able to observe some state of the system, whose confidentiality label is sufficiently low, and to modify some state of the system, whose integrity label is sufficiently low. Semantically, robust declassification says that if the attacker is unable to learn a secret with one attack, no other attack will cause it to be revealed [120, 177]. The attacker has no control over information release because all attacks are equally good. When applied to a decentralized system, robust declassification means that for any principal p, other principals that p does not trust cannot influence declassification of p's secrets [43].

To enforce robust declassification, prior security-typed languages place integrity constraints on declassification. The original work on FLAM enforces robust declassification using the voice operator ∇. However, when declassification is expressed as a programming-language operation, as is more typical, it is convenient to define a new operator on labels, one that maps in the other direction, from integrity to confidentiality. We define the *view* of a principal as the upper bound on the confidentiality a label or context can enforce to securely endorse that label:

Definition 3.1 (Principal view)**.** Let $\ell = p^{\rightarrow} \vee q^{\leftarrow}$ be a FLAM label (principal) expressed in normal form. The *view* of ℓ, written $\Delta(\ell)$, is defined as $\Delta(p^{\rightarrow} \vee q^{\leftarrow}) \triangleq q^{\rightarrow}$.

When the confidentiality of a label ℓ lies above the view of its own integrity, a declassification of that label may give adversaries the opportunity to subvert the declassification to release information. Without enough integrity, an adversary might, for example, replace the information that is intended to be released via declassification with some other secret.

Figure 3.3 illustrates this idea graphically. It depicts the lattice of FLAM labels, which is a product lattice with two axes, confidentiality and integrity. A given label ℓ is a point in this diagram, whereas the set of labels sharing the same confidentiality $\ell^{\rightarrow}$ or integrity $\ell^{\leftarrow}$ correspond to lines on the diagram. Given the integrity $\ell^{\leftarrow}$ of the label ℓ, the view of that integrity, $\Delta(\ell^{\leftarrow})$, defines a region of information (shaded) that is too confidential to be declassified.

The view operator directly corresponds to the writers-to-readers operator that Chong and Myers [43] use to enforce robust declassification in the DLM. We generalize the same idea here to the more expressive labels of FLAM.

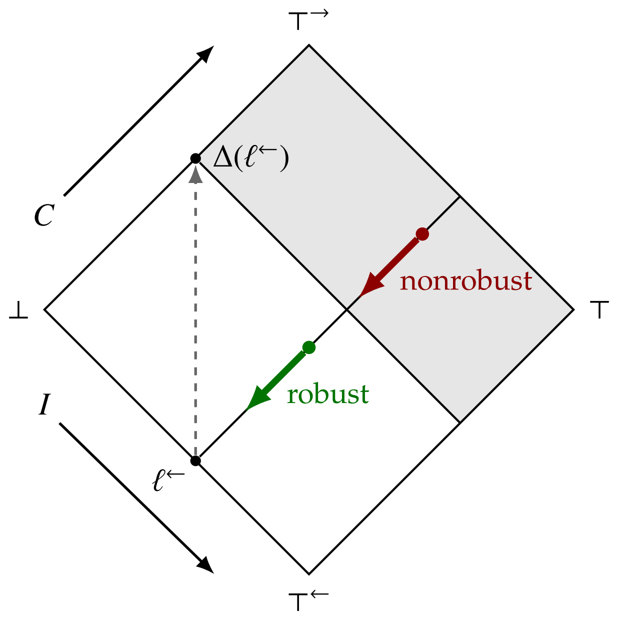


Figure 3.3: Robust declassification says information at level ℓ can be declassified only if it has enough integrity. The gray shaded region represents information that $\Delta(\ell^{\leftarrow})$ cannot read, so it is unsafe to declassify with ℓ's integrity.

3.3.2 Transparent Endorsement

The key insight of this work is that endorsement should be restricted in a manner dual to robust declassification; declassification (reducing confidentiality) requires a minimum integrity, so endorsement (raising integrity) should require a *maximum* confidentiality. Intuitively, if a principal could have written data it cannot read, which we call an "opaque write," it is unsafe to endorse that data. An endorsement is *transparent* if it endorses only information its authors could read.

The voice operator suffices to express this new restriction conveniently, as depicted in Figure 3.4. In the figure, we consider endorsing information with confidentiality $\ell^{\rightarrow}$. This confidentiality is mapped to a corresponding integrity level $\nabla(\ell^{\rightarrow})$, defining a minimal integrity level that ℓ must have in order to be endorsed. If ℓ lies below this boundary, its endorsement is considered transparent; if it lies above the boundary, endorsement is *opaque* and hence insecure. The duality with

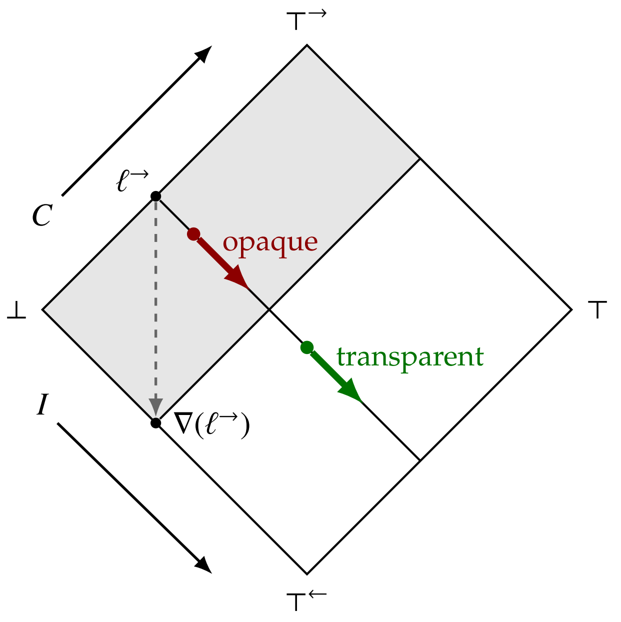


Figure 3.4: Transparent endorsement in NMIFC. The gray shaded region represents information that $\nabla(\ell^{\rightarrow})$ does not trust and may have been created by an opaque write. It is thus unsafe to endorse with ℓ's confidentiality.

robust declassification is clear.

3.4 A Core Language: NMIFC

We now describe the NonMalleable Information Flow Calculus (NMIFC), a new core language, modeled on DCC and FLAC, that allows downgrading, but in a more constrained manner than FLAC so as to provide stronger semantic guarantees. NMIFC incorporates the program-counter label *pc* of FLAC, but eschews the more powerful `assume` mechanism of FLAC in favor of more traditional declassify and endorse operations.

The full NMIFC is a small extension of Polymorphic DCC [1]. In Figure 3.5 we present the core syntax, leaving other features such as sums, pairs, and polymorphism to Section 3.9. Unlike DCC, NMIFC supports downgrading and models it as an effect. It is necessary to track what information influences control flow so

$$
\begin{array}{lll}
n \in \mathcal{N} & & \text{(atomic principals)} \\
x \in \mathcal{V} & & \text{(variable names)} \\
\pi \in \{\rightarrow, \leftarrow\} & & \text{(security aspects)} \\
\\
p, \ell, pc & ::= & n \mid \top \mid \bot \mid p^{\pi} \mid p \wedge p \mid p \vee p \mid p \sqcup p \mid p \sqcap p \\
\tau & ::= & \textsf{unit} \mid \tau \xrightarrow{pc} \tau \mid \ell \ \textsf{says}\ \tau \\
v & ::= & () \mid \lambda x{:}\tau[pc].\, e \mid (\overline{\eta}_{\ell}\ v) \\
e & ::= & x \mid v \mid e\ e \mid (\eta_{\ell}\ e) \mid \textsf{bind}\ x = e\ \textsf{in}\ e \\
 & \mid & \textsf{decl}\ e\ \textsf{to}\ \ell \mid \textsf{endorse}\ e\ \textsf{to}\ \ell
\end{array}
$$

Figure 3.5: Core NMIFC syntax.

that these downgrading effects may be appropriately constrained. Therefore, like FLAC, NMIFC adds *pc* labels to lambda terms and types.

Similarly to DCC, protected values have type ℓ says τ where ℓ is the confidentiality and integrity of a value of type τ. All computation on these values occurs in the says monad; protected values must be bound using the bind term before performing operations on them (e.g., applying them as functions). Results of such computations are protected with the monadic unit operator $(\eta_{\ell}\ e)$, which protects the result of e with label ℓ.

3.4.1 NMIFC Operational Semantics

The core semantics of NMIFC are mostly standard, but to obtain our theoretical results we need additional information about evaluation. This information is necessary because we want to identify, for instance, whether information is ever available to an attacker during evaluation, even if it is discarded and does not influence the final result. This approach gives an attacker more power; an attacker can see information at its level even if it is not output by the program.

The NMIFC semantics, presented in Figure 3.6, maintain a trace t of events.

$\boxed{e \longrightarrow e'}$

[E-APP] $$\frac{}{(\lambda\, x\!:\!\tau[pc].\, e)\; v \longrightarrow e[x \mapsto v]}$$

[E-BINDM] $$\frac{}{\text{bind } x = (\overline{\eta}_\ell\; v) \text{ in } e \longrightarrow e[x \mapsto v]}$$

$$\begin{array}{rl} \text{(event)} & c ::= \bullet \mid (\overline{\eta}_\ell\; v) \mid (\downarrow^{\pi}_{\ell'},\, \overline{\eta}_\ell\; v) \\ \text{(trace)} & t ::= \varepsilon \mid c \mid t; t \end{array}$$

$\boxed{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t'\rangle}$

[E-STEP] $$\frac{e \longrightarrow e'}{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t; \bullet\rangle}$$

[E-UNITM] $$\frac{}{\langle (\eta_\ell\; v),\, t\rangle \twoheadrightarrow \langle (\overline{\eta}_\ell\; v),\, t; (\overline{\eta}_\ell\; v)\rangle}$$

[E-DECL] $$\frac{}{\langle \text{decl } (\overline{\eta}_{\ell'}\; v) \text{ to } \ell,\, t\rangle \twoheadrightarrow \left\langle (\overline{\eta}_\ell\; v),\, t; (\downarrow^{\rightarrow}_{\ell'},\, \overline{\eta}_\ell\; v)\right\rangle}$$

[E-ENDORSE] $$\frac{}{\langle \text{endorse } (\overline{\eta}_{\ell'}\; v) \text{ to } \ell,\, t\rangle \twoheadrightarrow \left\langle (\overline{\eta}_\ell\; v),\, t; (\downarrow^{\leftarrow}_{\ell'},\, \overline{\eta}_\ell\; v)\right\rangle}$$

[E-EVAL] $$\frac{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t'\rangle}{\langle E[e],\, t\rangle \twoheadrightarrow \langle E[e'],\, t'\rangle}$$

Evaluation context

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\; e \mid v\; E \mid (\eta_\ell\; E) \mid \text{bind } x = E \text{ in } e \\ & \mid & \text{decl } E \text{ to } \ell \mid \text{endorse } E \text{ to } \ell \end{array}$$

Figure 3.6: Core NMIFC operational semantics.

An event is emitted into the trace whenever a new protected value is created and whenever a declassification or endorsement occurs. These events track the observations or influence an attacker may have during a run of an NMIFC program. Formally, a trace can be an empty trace ε, a single event c, or the concatenation of two traces with the associative operator ";" with identity ε.

When a source-level unit term $(\eta_\ell\ v)$ is evaluated (rule E-UNITM), an event $(\overline{\eta}_\ell\ v)$ is added to the trace indicating that the value v became protected at ℓ. When a protected value is declassified, a declassification event $(\downarrow^{\rightarrow}_{\ell'}, \overline{\eta}_\ell\ v)$ is emitted, indicating that v was declassified from ℓ' to ℓ. Likewise, an endorsement event $(\downarrow^{\leftarrow}_{\ell'}, \overline{\eta}_\ell\ v)$ is emitted for an endorsement. Other evaluation steps (rule E-STEP) emit $\bullet$, for "no event." Rule E-EVAL steps under the evaluation contexts [174] defined at the bottom of Figure 3.6.

Rather than being literal side effects of the program, these events track how observable information is as it is accessed, processed, and protected by the program. Because our semantics emits an event whenever information is protected (by evaluating an η term) or downgraded (by a decl or endorse term), our traces capture all information processed by a program, indexed by the policy protecting that information.

By analogy, these events are similar to the typed and labeled mutable reference cells of languages like FlowCaml [133] and DynSec [184]. An event $(\overline{\eta}_\ell\ v)$ is analogous to allocating a reference cell protected at ℓ, and $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_\ell\ v)$ is analogous to copying the contents of a cell at ℓ' to a new cell at ℓ.

It is important for the semantics to keep track of these events so that our security conditions hold for programs containing data structures and higher-order functions. Previous language-based definitions of robust declassification have only applied to simple while-languages [12, 43, 120] or to primitive types [9].

$\boxed{\ell \triangleleft \tau}$

$$[\text{P-UNIT}]\ \frac{}{\ell \triangleleft \mathsf{unit}} \qquad\qquad [\text{P-LBL}]\ \frac{\ell' \sqsubseteq \ell}{\ell' \triangleleft \ell \ \mathsf{says}\ \tau}$$

Figure 3.7: Type protection levels.

3.4.2 NMIFC Type System

The NMIFC protection relation (Figure 3.7) defines how types relate to information flow policies. A type τ protects the confidentiality and integrity of ℓ if $\ell \triangleleft \tau$. Unlike in DCC and FLAC, a label is protected by a type only if it flows to the outermost says principal. In FLAC and DCC, the types ℓ' says ℓ says τ and ℓ says ℓ' says τ protect the same set of principals; in other words, says is commutative. By distinguishing between these types, NMIFC does not provide the same commutativity.

The commutativity of says is a design decision, offering a more permissive programming model at the cost of less precise tracking of dependencies. NMIFC takes advantage of this extra precision in the UNITM typing rule so the label on every η term protects the information it contains, even if nested within other η terms. Abadi [2] similarly modifies DCC's protection relation to distinguish the protection level of terms with nested says types.

The core type system presented in Figure 3.8 enforces nonmalleable information flow for NMIFC programs. Most of the typing rules are standard, and differ only superficially from DCC and FLAC. Like in FLAC, NMIFC typing judgments include a program counter label, *pc*, that represents an upper bound on the confidentiality and integrity of bound information that any computation may depend upon. For instance, rule BINDM requires the type of the body of a bind term to protect the unlabeled information of type τ' with at least ℓ, and to type-check under a raised program counter label $pc \sqcup \ell$. Rule LAM ensures that function bodies type-

$\boxed{\Gamma; pc \vdash e : \tau}$

$$[\text{VAR}]\ \frac{\Gamma(x) = \tau}{\Gamma; pc \vdash x : \tau} \qquad [\text{UNIT}]\ \frac{}{\Gamma; pc \vdash () : \mathsf{unit}} \qquad [\text{LAM}]\ \frac{\Gamma, x\!:\!\tau_1; pc' \vdash e : \tau_2}{\Gamma; pc \vdash \lambda x\!:\!\tau_1[pc'].\, e : \tau_1 \xrightarrow{pc'} \tau_2}$$

$$[\text{APP}]\ \frac{\Gamma; pc \vdash e_1 : \tau' \xrightarrow{pc'} \tau \qquad \Gamma; pc \vdash e_2 : \tau' \qquad pc \sqsubseteq pc'}{\Gamma; pc \vdash e_1\ e_2 : \tau} \qquad [\text{UNITM}]\ \frac{\Gamma; pc \vdash e : \tau \qquad pc \sqsubseteq \ell}{\Gamma; pc \vdash (\eta_\ell\ e) : \ell\ \mathsf{says}\ \tau}$$

$$[\text{VUNITM}]\ \frac{\Gamma; pc \vdash v : \tau}{\Gamma; pc \vdash (\overline{\eta}_\ell\ v) : \ell\ \mathsf{says}\ \tau} \qquad [\text{BINDM}]\ \frac{\Gamma; pc \vdash e : \ell\ \mathsf{says}\ \tau' \qquad \ell \triangleleft \tau \qquad \Gamma, x\!:\!\tau'; pc \sqcup \ell \vdash e' : \tau}{\Gamma; pc \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \tau}$$

$$[\text{DECL}]\ \frac{\Gamma; pc \vdash e : \ell'\ \mathsf{says}\ \tau \qquad pc \sqsubseteq \ell \qquad \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow}) \qquad \ell'^{\leftarrow} = \ell^{\leftarrow}}{\Gamma; pc \vdash \mathsf{decl}\ e\ \mathsf{to}\ \ell : \ell\ \mathsf{says}\ \tau}$$

$$[\text{ENDORSE}]\ \frac{\Gamma; pc \vdash e : \ell'\ \mathsf{says}\ \tau \qquad pc \sqsubseteq \ell \qquad \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow}) \qquad \ell'^{\rightarrow} = \ell^{\rightarrow}}{\Gamma; pc \vdash \mathsf{endorse}\ e\ \mathsf{to}\ \ell : \ell\ \mathsf{says}\ \tau}$$

Figure 3.8: Typing rules for core NMIFC.

check with respect to the function's *pc* annotation, and rule APP ensures functions are only applied in contexts that flow to these annotations.

The NMIFC rule for UNITM differs from FLAC and DCC in requiring the premise $pc \sqsubseteq \ell$ for well-typed η terms. This premise ensures a more precise relationship between the *pc* and η terms. Intuitively this restriction makes sense. The *pc* is a bound on all unlabeled information in the context. Since an expression e protected with $(\eta_\ell\ e)$ may depend on any of this information, it makes sense to require that *pc* flow to ℓ.[2]

By itself, this restrictive premise would prevent any public data from flowing through secret contexts and trusted data from flowing through untrusted contexts.

[2]The premise is not required in FLAC because protection is commutative. For example, in a FLAC term such as $\mathsf{bind}\ x = v\ \mathsf{in}\ (\eta_{\ell'}\ (\eta_\ell\ x))$, x may be protected by ℓ or ℓ'.

To allow such flows, we distinguish source-level ($\eta_\ell\ e$) terms from run-time values ($\overline{\eta}_\ell\ v$), which have been fully evaluated. These terms are only created by the operational semantics during evaluation and no longer depend on the context in which they appear; they are closed terms. Thus it is appropriate to omit the new premise in VUnitM. This approach allows us to require more precise flow tracking for the explicit dependencies of protected expressions without restricting where these values flow once they are fully evaluated.

Rule Decl ensures a declassification from label ℓ' to ℓ is robust. We first require $\ell'^{\leftarrow} = \ell^{\leftarrow}$ to ensure that this does not perform endorsement. A more permissive premise $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow}$ is admissible, but requiring equality simplifies our proofs and does not reduce expressiveness since the declassification can be followed by a subsequent relabeling. The premise $pc \sqsubseteq \ell$ requires that declassifications occur in high-integrity contexts, and prevents declassification events from creating implicit flows. The premise $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})$ ensures that the confidentiality of the information declassified does not exceed the view of the integrity of the principals that may have influenced it. These influences can be either explicit ($\ell'^{\leftarrow}$) or implicit ($pc^{\leftarrow}$), so we compare against the join of the two.[3] This last premise effectively combines the two conditions identified by Chong and Myers [43] for enforcing robust declassification in an imperative while-language.

Rule Endorse enforces transparent endorsement. All but the last premise are straightforward: the expression does not declassify and $pc \sqsubseteq \ell$ requires a high-integrity context to endorse and prevents implicit flows. Interestingly, the last premise is dual to that in Decl. An endorsement cannot raise integrity above the voice of the confidentiality of the data being endorsed ($\ell'^{\rightarrow}$) or the context performing the endorsement ($pc^{\rightarrow}$). As in Decl, we compare against their join.

[3] The first two premises—$\ell'^{\leftarrow} = \ell^{\leftarrow}$ and $pc \sqsubseteq \ell$—make this join redundant. It would, however, be necessary if we replaced the equality premise with the more permissive $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow}$ version, so we include it for clarity.

```
checkpwd = $\lambda g : U^{\leftarrow}$ says String, $p : T$ says String[$T^{\leftarrow}$].
  bind $guess$ = (endorse $g$ to $T^{\leftarrow}$) in
    decl (bind $pwd = p$ in ($\eta_T$ $pwd == guess$)) to $T^{\leftarrow}$
```

Figure 3.9: A secure version of a password checker.

3.4.3 Examples Revisited

We now reexamine the examples presented in Section 3.1 to see that the NMIFC type system prevents the vulnerabilities seen above.

Password Checker

We saw above that when the password checker labels guess at $T^{\rightarrow}$, well-typed code can improperly set guess to the actual password. We noted that the endorsement enabled an insecure flow of information. Looking at ENDORSE in NMIFC, we can attempt to type the equivalent expression: endorse $guess$ to T. However, if $guess$ has type $T^{\rightarrow}$ says bool, the endorse does not type-check; it fails to satisfy the final premise of ENDORSE:

$$\bot^{\leftarrow} = (T^{\rightarrow})^{\leftarrow} \not\sqsubseteq T^{\leftarrow} \sqcup \nabla(T^{\rightarrow}) = T^{\leftarrow}.$$

If we instead give guess the label $U^{\leftarrow}$, the endorsement type-checks, assuming a sufficiently trusted pc.

This is as it should be. With the label $U^{\leftarrow}$, the guesser must be able to read their own guess, guaranteeing that they cannot guess the correct password unless they in fact know the correct password. Figure 3.9 shows this secure version of the password checker.

Sealed-Bid Auction

In the insecure auction described in Section 3.1.2, we argued that an insecure flow was created when T endorsed `b_bid` from $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}$ to $A \wedge B$. This endorsement requires a term of the form $\textsf{endorse}\ v\ \textsf{to}\ A \wedge B$ where v types to $B^{\leftarrow} \wedge (A \wedge B)^{\rightarrow}\ \textsf{says int}$. Despite the trusted context, the last premise of ENDORSE again fails:

$$B^{\leftarrow} \not\sqsubseteq (A \wedge B)^{\leftarrow} \sqcup \nabla((A \wedge B)^{\rightarrow}) = (A \wedge B)^{\leftarrow}.$$

If we instead label $\textsf{a\_bid} : A\ \textsf{says int}$ and $\textsf{b\_bid} : B\ \textsf{says int}$, then the corresponding $\textsf{endorse}$ statements type-check, assuming that T is trusted: $T \sqsubseteq (A \wedge B)^{\leftarrow}$.

Laundering Secrets

For the secret-laundering example in Section 3.1.3, we assume that neither H nor L is trusted, but the output from the program is. This forces an implicit endorsement of y, the input received from H. But the condition needed to endorse from $H^{\rightarrow} \wedge \bot^{\leftarrow}$ to $H^{\rightarrow} \wedge \top^{\leftarrow}$ is false:

$$\bot^{\leftarrow} \sqsubseteq \top^{\leftarrow} \sqcup \nabla(H^{\rightarrow}) = \nabla(H^{\rightarrow})$$

We have $\nabla(L^{\rightarrow}) \not\sqsubseteq \nabla(H^{\rightarrow})$ and all integrity flows to $\bot^{\leftarrow}$, so by transitivity the above condition cannot hold.

3.5 Security Conditions

The NMIFC typing rules enforce several strong security conditions: multiple forms of conditional noninterference, robust declassification, and our new transparent endorsement and nonmalleable information flow conditions. We define these conditions formally but relegate proof details to Section 3.12.

3.5.1 Attackers

Noninterference is frequently stated with respect to a specific but arbitrary label. Anything below that label in the lattice is "low" (public or trusted) and everything else is "high". We broaden this definition slightly and designate high information using a set of labels $\mathcal{H}$ that is upward closed. That is, if $\ell \in \mathcal{H}$ and $\ell \sqsubseteq \ell'$, then $\ell' \in \mathcal{H}$. We refer to such upward closed sets as *high sets*.

We say that a type τ is a *high type*, written "$\vdash \tau \;\mathsf{prot}\; \mathcal{H}$", if all of the information in a value of type τ is above some label in the high set $\mathcal{H}$. The following rule defines high types:

$$[\text{P-S}\textsc{et}]\;\frac{H \in \mathcal{H} \qquad H \triangleleft \tau}{\vdash \tau \;\mathsf{prot}\; \mathcal{H}}\;\mathcal{H} \text{ is upward closed}$$

This formulation of adversarial power is adequate to express noninterference, in which confidentiality and integrity do not interact. However, our more complex conditions relate confidentiality to integrity and therefore require a way to relate the attacker's power in the two domains.

Intuitively, an attacker is an arbitrary set of colluding atomic principals. Specifically, if the attacker controls atomic principals $n_1, \ldots, n_k \in \mathcal{N}$, then the set of labels $\mathcal{A} = \{\ell \in \mathcal{L} \mid n_1 \wedge \cdots \wedge n_k \Rightarrow \ell\}$ represents the attacker's power. These principals may include principals mentioned in the program, and there may be delegations between attacker principals and program principals. While this definition of an attacker is intuitive, the results in this paper actually hold for a more general notion of attacker defined in Section 3.10.

Attackers correspond to two high sets: an *untrusted* set $\mathcal{U} = \{\ell \in \mathcal{L} \mid \ell^{\leftarrow} \in \mathcal{A}\}$ and a *secret* set $\mathcal{S} = \{\ell \in \mathcal{L} \mid \ell^{\rightarrow} \notin \mathcal{A}\}$. We say that $\mathcal{A}$ *induces* $\mathcal{U}$ and $\mathcal{S}$.

$\boxed{c \approx_{\mathcal{W}} c'}$ $\boxed{v \approx_{\mathcal{W}} v'}$

These equivalence relations are the smallest congruences over c and over v extended with $\bullet$, containing the equivalences defined by these rules:

$$[\text{Eq-UnitM}]\ \frac{\ell \notin \mathcal{W}}{(\overline{\eta}_\ell\ v) \approx_{\mathcal{W}} \bullet} \qquad [\text{Eq-Down}]\ \frac{\ell \notin \mathcal{W}}{(\downarrow^\pi_{\ell'}, \overline{\eta}_\ell\ v) \approx_{\mathcal{W}} \bullet}$$

$\boxed{t \approx^\star_{\mathcal{W}} t'}$

The equivalence relation $t \approx^\star_{\mathcal{W}} t'$ is the smallest congruence over t containing the equivalences defined by these rules:

$$[\text{T-Lift}]\ \frac{c \approx_{\mathcal{W}} c'}{c \approx^\star_{\mathcal{W}} c'} \qquad [\text{T-BulletR}]\ t; \bullet \approx^\star_{\mathcal{W}} t \qquad [\text{T-BulletL}]\ \bullet; t \approx^\star_{\mathcal{W}} t$$

Figure 3.10: Low equivalence and low trace equivalence.

3.5.2 Equivalences

All of our security conditions involve relations on traces. As is typically the case for information-flow security conditions, we define a notion of "low equivalence" on traces, which ignores effects with high labels. We proceed by defining low-equivalent expressions and then extending low-equivalence to traces.

For expression equivalence, we examine precisely the values which are visible to a low observer defined by a set of labels $\mathcal{W}$: $(\overline{\eta}_\ell\ v)$ and $(\downarrow^\pi_{\ell'}, \overline{\eta}_\ell\ v)$ where $\ell \in \mathcal{W}$. We formalize this idea in Figure 3.10, using $\bullet$ to represent values that are not visible. Beyond ignoring values unable to affect the output, we use a standard structural congruence (i.e., syntactic equivalence). This strict notion of equivalence is not entirely necessary; observational equivalence or any refinement thereof would be sufficient if augmented with the $\bullet$-equivalences in Figure 3.10.

Figure 3.10 also extends the equivalence on emitted values to equivalence on entire traces of emitted values. Essentially, two traces are equivalent if there is a way to match up equivalent events in each trace, while ignoring high events equivalent to $\bullet$.

3.5.3 Noninterference and Downgrading

An immediate consideration when formalizing information flow is how to express interactions between an adversary and the system. One possibility is to limit interaction to inputs and outputs of the program. This is a common approach for functional languages. We take a stronger approach in which security is expressed in terms of execution traces. Note that traces contain all information necessary to ensure the security of input and output values.

We begin with a statement of noninterference in the presence of downgrading. Theorem 3.1 states that, given two high inputs, a well-typed program produces two traces that are either low-equivalent or contain a downgrade event that distinguishes them. This implies that differences in traces distinguishable by an attacker are all attributable to downgrades of information derived from the high inputs. Furthermore, any program that performs no downgrades on secret or untrusted values (i.e., contain no decl or endorse terms on $\mathcal{H}$ data) must be noninterfering.

Theorem 3.1 (Noninterference modulo downgrading)**.** *Let $\mathcal{H}$ be a high set and let $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given an expression e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if*

$$\langle e[x \mapsto v_i],\ v_i\rangle \longrightarrow\!\!\!\!\!\rightarrow^{*} \left\langle v_i',\ t^i\right\rangle$$

then either there is some event $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w) \in t^i$ where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, or $t^1 \approx^{\star}_{\mathcal{W}} t^2$.

The restrictions placed on downgrading operations mean that we can characterize the conditions under which no downgrading can occur. We add two further noninterference theorems that restrict downgrading in different ways. Theorem 3.2

states that if a program types without a public–trusted *pc* it must be noninterfering (with respect to that definition of "public–trusted").

Theorem 3.2 (Noninterference of high-*pc* programs)**.** *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H}$ be one of those high sets and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given some e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1 \;\mathsf{prot}\; \mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\, v_i\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle v_i',\, t^i\rangle$ and $pc \in \mathcal{U} \cup \mathcal{S}$, then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

Rather than restrict the *pc*, Theorem 3.3 states that secret–untrusted information is *always* noninterfering. Previous work [e.g., 12, 120] does not restrict endorsement of confidential information, allowing any label to be downgraded to public–trusted (given a public–trusted *pc*). In NMIFC, however, secret–untrusted data must remain secret and untrusted.

Theorem 3.3 (Noninterference of secret–untrusted data)**.** *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H} = \mathcal{U} \cap \mathcal{S}$ and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given some e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1 \;\mathsf{prot}\; \mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\, v_i\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle v_i',\, t^i\rangle$ then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

3.5.4 Robust Declassification and Irrelevant Inputs

We now move to security conditions for programs that do not satisfy noninterference. Recall that robust declassification informally means the attacker has no influence on what information is released by declassification. Traditionally, it is stated in terms of attacker-provided code that is inserted into low-integrity holes in programs which differ only in their secret inputs. In NMIFC, the same attacker power can be obtained by substituting exactly two input values into the program, one secret and one untrusted. This simplification is possible because NMIFC has first-class functions that can model the substitution of low-integrity code. Section 3.11

shows that this simpler two-input definition is equivalent to the traditional hole-based approach in the full version of NMIFC (Section 3.9).

Prior work on while-based languages [43, 120] defines robust declassification in terms of four traces generated by the combination of two variations: a secret input and some attacker-supplied code. For terminating traces, these definitions require any pair of secrets to produce public-equivalent traces under all attacks or otherwise to produce distinguishable traces regardless of the attacks chosen. This implies that an attacker cannot control the disclosure of secrets.

We can attempt to capture this notion of robust declassifcation using the notation of NMIFC. For a program e with a secret input x and untrusted input y, we wish to say e robustly declassifies if, for all secret values v_1, v_2 and for all untrusted values w_1, w_2, where

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\, v_i; w_j \right\rangle \longrightarrow\!\!\!\!\!\rightarrow^{*} \left\langle v_{ij},\, t^{ij} \right\rangle,$$

then $t^{11} \approx^{\star}_{\mathcal{P}} t^{21} \iff t^{12} \approx^{\star}_{\mathcal{P}} t^{22}$.

This condition is intuitive but, unfortunately, overly restrictive. It does not account for the possibility of an *inept attack*, in which an attacker causes a program to reveal less information than intended.

Inept attacks are harder to characterize than in previous work because, unlike the previously used while-languages, NMIFC supports data structures with heterogeneous labels. Using such data structures, we can build a program that implicitly declassifies data by using a secret to influence the selection of an attacker-provided value and then declassifying that selection. Figure 3.11 provides an example of such a program, which uses sums and products from the full NMIFC language.

While this program appears secure—the attacker has no control over what information is declassified or when a declassification occurs—it violates the above

```
(λ x:(P^→ ∧ U^← says τ) × (P^→ ∧ U^← says τ)[P^→ ∧ T^←].
   decl (bind b = (η_{S^→∧T^←} sec) in
           match b with
            | in1(_). (η_{S^→∧T^←} (proj1 x))
            | in2(_). (η_{S^→∧T^←} (proj2 x))
           end)
       to P^→ ∧ T^←) ⟨atk1, atk2⟩
```

Figure 3.11: A program that admits inept attacks. Here $P \sqsubseteq S$ and $T \sqsubseteq U$, but not vice versa, so *sec* is a secret boolean and $\langle atk_1, atk_2 \rangle$ form an untrusted pair of values. If $atk_1 \neq atk_2$, then the attacker will learn the value of *sec*. If $atk_1 = atk_2$, however, then the attacker learns nothing due to its own ineptness.

condition. One attack can contain the same value twice—causing any two secrets to produce indistinguishable traces—while the other can contain different values. Intuitively, no vulnerability in the program is thereby revealed; the program was *intended* to release information, but the attacker failed to infer it due to a poor choice of attack. Such inputs result in less information leakage entirely due to the attacker's ineptness, not an insecurity of the program. As a result, we consider inputs from inept attackers to be *irrelevant* to our security conditions.

Dually to inept attackers, we can define uninteresting secret inputs. For example, if a program endorses an attacker's selection of a secret value, an input where all secret options contain the same data is uninteresting, so we also consider it irrelevant.

Which inputs are irrelevant is specific to the program and to the choice of attacker. In Figure 3.11, if both execution paths used ($\mathsf{proj}_1\ x$), there would be no way for an attacker to learn any information, so all attacks are equally relevant. Similarly, if $S^{\rightarrow}$ is already considered public, then there is no secret information in the first place, so again, all attacks are equally relevant.

For an input to be irrelevant, it must have no influence over the outermost layer of the data structure—the label that is explicitly downgraded. If the input could influence that outer layer in any way, the internal data could be an integral part of an insecure execution. Conversely, when the selection of nested values is independent of any untrusted/secret information (though the content of the values may not be), it is reasonable to assume that the inputs will be selected so that different choices yield different results. An input which does not is either an inept attack—an attacker gaining less information than it could have—or an uninteresting secret—a choice between secrets that are actually the same. In either case, the input is irrelevant.

To ensure that we only consider data structures with nested values that were selected independently of the values themselves, we leverage the noninterference theorems in Section 3.5.3. In particular, if the outermost label is trusted before a declassification (or public prior to an endorsement), then any influence from untrusted (secret) data must be the result of a prior explicit downgrade. Thus we can identify irrelevant inputs by finding inputs that result in traces that are public-trusted equivalent, but can be made both public (trusted) equivalent and non-equivalent at the point of declassification (endorsement).

To define this formally, we begin by partitioning the principal lattice into four quadrants using the definition of an attacker from Section 3.5.1. We consider only flows between quadrants and, as with noninterference, downgrades must result in public or trusted values. We additionally need to refer to individual elements and prefixes of traces. For a trace t, let t_n denote the nth element of t, and let $t_{..n}$ denote the prefix of t containing its first n elements.

Definition 3.2 (Irrelevant inputs)**.** Consider attacker $\mathcal{A}$ inducing high sets $\mathcal{H}_{\leftarrow}$ and $\mathcal{H}_{\rightarrow}$. Let $\mathcal{W}_{\pi} = \mathcal{L} \setminus \mathcal{H}_{\pi}$ and $\mathcal{W} = \mathcal{W}_{\leftarrow} \cap \mathcal{W}_{\rightarrow}$. Given opposite projections π and π', a program e, and types τ_x and τ_y such that $\vdash \tau_x \text{ prot } \mathcal{H}_{\pi'}$ and $\vdash \tau_y \text{ prot } \mathcal{H}_{\pi}$, we say an input v_1 is an *irrelevant π'-input* with respect to $\mathcal{A}$ and e if $\Gamma; pc \vdash v_1 : \tau_x$ and there exist values v_2, w_1, and w_2 and four trace indices n_{ij} (for $i, j \in \{1, 2\}$) such that the following conditions hold:

1. $\Gamma; pc \vdash v_2 : \tau_x$, $\Gamma; pc \vdash w_1 : \tau_y$, and $\Gamma; pc \vdash w_2 : \tau_y$
2. $\langle e[x \mapsto v_i][y \mapsto w_j],\, v_i; w_j\rangle \longrightarrow\!\!\!\!\rightarrow^* \langle v_{ij},\, t^{ij}\rangle$
3. $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$ for all $i, j \in \{1, 2\}$
4. $t^{ij}_{..n_{ij}} \approx^{\star}_{\mathcal{W}} t^{kl}_{..n_{kl}}$ for all $i, j, k, l \in \{1, 2\}$
5. $t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{W}_\pi} t^{12}_{..n_{12}}$
6. $t^{21}_{..n_{21}} \not\approx_{\ell}\, \mathcal{W}_\pi{}^{\star} t^{22}_{..n_{22}}$

Otherwise we say v_1 is a *relevant π'-input* with respect to $\mathcal{A}$ and e, denoted $\mathrm{rel}^{\pi'}_{\mathcal{A},e}(v_1)$. Note that the four indices n_{ij} identify corresponding prefixes of the four traces.

As mentioned above, prior downgrades can allow secret/untrusted information to directly influence the outer later of the data structure, but Condition 4 requires that all four trace prefixes be public-trusted equivalent, so any such downgrades must have the same influence across all executions. Condition 5 requires that some inputs result in prefixes that are public equivalent (or trusted equivalent for endorsement), while Condition 6 requires that other inputs result in prefixes that are distinguishable. Since all prefixes are public-trusted equivalent, this means there is an implicit downgrade inside a data structure, so the equivalent prefixes form an irrelevant input.

We can now relax our definition of robust declassification to only restrict the behavior of *relevant* inputs.

Definition 3.3 (Robust declassification)**.** Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that *e robustly declassifies* if, for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$ (and $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$) and all values v_1, v_2, w_1, w_2, if

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j \right\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \left\langle v_{ij},\ t^{ij} \right\rangle,$$

then $\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1) \text{ and } t^{11} \approx^{\star}_{\mathcal{P}} t^{21}\right) \implies t^{12} \approx^{\star}_{\mathcal{P}} t^{22}$.

As NMIFC only restricts declassification of low-integrity data, endorsed data is free to influence future declassifications. As a result, we can only guarantee robust declassification in the absence of endorsements.

Theorem 3.4 (Robust declassification)**.** *Given a program e, if $\Gamma, x{:}\tau_x, y{:}\tau_y; pc \vdash e : \tau$ and e contains no* endorse *expressions, then e robustly declassifies.*

Note that prior definitions of robust declassification [43, 120] similarly prohibit endorsement and ignore pathological inputs, specifically nonterminating traces. Our irrelevant inputs are very different since NMIFC is strongly normalizing but admits complex data structures, but the need for some restriction is not new.

3.5.5 Transparent Endorsement

We described in Section 3.1 how endorsing opaque writes can create security vulnerabilities. To formalize this intuition, we present *transparent endorsement*, a security condition that is dual to robust declassification. Instead of ensuring that untrusted information cannot meaningfully influence declassification, transparent endorsement guarantees that secret information cannot meaningfully influence endorsement. This guarantee ensures that secrets cannot influence the endorsement of an attacker's value—neither the value endorsed nor the existence of the endorsement itself.

As it is completely dual to robust declassification, we again appeal to the notion of irrelevant inputs, this time to rule out uninteresting secrets. The condition looks nearly identical, merely switching the roles of confidentiality and integrity. It therefore ensures that any choice of interesting secret provides an attacker with the maximum possible ability to influence endorsed values; no interesting secrets provide more power to attackers than others.

Definition 3.4 (Transparent endorsement)**.** Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that *e transparently endorses* if, for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$ (and $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$) and all values v_1, v_2, w_1, w_2, if

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j \right\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \left\langle v_{ij},\ t^{ij} \right\rangle,$$

then $\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11} \approx^{\star}_{\mathcal{T}} t^{12}\right) \implies t^{21} \approx^{\star}_{\mathcal{T}} t^{22}$.

As in robust declassification, we can only guarantee transparent endorsement in the absence of declassification.

Theorem 3.5 (Transparent endorsement)**.** *Given a program e, if $\Gamma, x{:}\tau_x, y{:}\tau_y; pc \vdash e : \tau$ and e contains no* decl *expressions, then e transparently endorses.*

3.5.6 Nonmalleable Information Flow

Robust declassification restricts declassification and transparent endorsement restricts endorsement, but as structured above, neither cannot be enforced in the presence of both declassification and endorsement. The key difficulty stems from the fact that previously declassified and endorsed data should be able to influence future declassifications and endorsements. However, any endorsement allows an attack to influence declassification, so varying the secret input can cause the traces to deviate for one attack and not another. Similarly, once a declassification has occurred, we can say little about the relation between trace pairs that fix a secret and vary an attack.

There is one condition that allows us to safely relate trace pairs even after a downgrade event: if the downgraded values are identical in both trace pairs. Even if a declassify or endorse could have caused the traces to deviate, if it did not, then this program is essentially the same as one that started with that value already downgraded and performed no downgrade. To capture this intuition, we define nonmalleable information flow in terms of trace prefixes that either do not deviate in public values when varying only the secret input or do not deviate in trusted values when varying only the untrusted input. This assumption may seem strong at first, but it exactly captures the intuition that downgraded data—but not secret/untrusted data—should be able to influence future downgrades. While two different endorsed attacks could influence a future declassification, if the attacks are similar enough to result in the same value being endorsed, they must influence the declassification *in the same way*.

Definition 3.5 (Nonmalleable information flow)**.** Let e be a program and let x and y be variables representing secret and untrusted inputs, respectively. We say that e enforces *nonmalleable information flow* (NMIF) if the following holds for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$, $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$ and $\mathcal{W} = \mathcal{T} \cap \mathcal{P}$. For all values v_1, v_2, w_1, and w_2, let

$$\left\langle e[x \mapsto v_i][y \mapsto w_j],\ v_i; w_j \right\rangle \longrightarrow\!\!\!\!\!\rightarrow^{*} \left\langle v_{ij},\ t^{ij} \right\rangle.$$

For all indices n_{ij} such that $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$

1. If $t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ for $i = 1, 2$, then

$$\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{P}} t^{21}_{..n_{21}}\right) \implies t^{12}_{..n_{12}} \approx^{\star}_{\mathcal{P}} t^{22}_{..n_{22}}.$$

2. Similarly, if $t^{1j}_{..n_{1j}-1} \approx^{\star}_{\mathcal{P}} t^{2j}_{..n_{2j}-1}$ for $j = 1, 2$, then

$$\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{T}} t^{12}_{..n_{12}}\right) \implies t^{21}_{..n_{21}} \approx^{\star}_{\mathcal{T}} t^{22}_{..n_{22}}.$$

Unlike the previous conditions, NMIFC enforces NMIF with no syntactic restrictions.

Theorem 3.6 (Nonmalleable information flow)**.** *For any program e such that $\Gamma, x\!:\!\tau_x, y\!:\!\tau_y; pc \vdash e : \tau$, e enforces NMIF.*

We note that both Theorems 3.4 and 3.5 are directly implied by Theorem 3.6. For robust declassification, the syntactic prohibition on endorse directly enforces $t^{i1} \approx^{\star}_{\mathcal{T}} t^{i2}$ (for the entire trace), and the rest of case 1 is exactly that of Theorem 3.4. Similarly, the syntactic prohibition on decl enforces $t^{1j} \approx^{\star}_{\mathcal{P}} t^{2j}$, while the rest of case 2 is exactly Theorem 3.5.

3.6 NMIF as 4-safety

Clarkson and Schneider [46] define a *hyperproperty* as "a set of sets of infinite traces," and *hypersafety* to be a hyperproperty that can be characterized by a finite set of finite trace prefixes defining some "bad thing." That is, given any of these finite sets of trace prefixes it is impossible to extend those traces to satisfy the hyperproperty. It is therefore possible to show that a program satisfies a hypersafety property by proving that no set of finite trace prefixes emitted by the program fall into this set of "bad things." They further define a *k-safety hyperproperty* (or *k-safety*) as a hypersafety property that limits the set of traces needed to identify a violation to size k.

Clarkson and Schneider note that noninterference is 2-safety. We demonstrate here that robust declassification, transparent endorsement, and nonmalleable information flow are all 4-safety properties.[4]

[4]While NMIFC produces finite traces and hyperproperties are defined for infinite traces, we can easily extend NMIFC traces by stuttering $\bullet$ infinitely after termination.

For a condition to be 2-safety, it must be possible to demonstrate a violation using only two finite traces. With noninterference, this demonstration is simple: if two traces with low-equivalent inputs are distinguishable by a low observer, the program is interfering.

Robust declassification, however, has no such representation. It says that the program's confidentiality release events cannot be influence by untrusted inputs. If we could precisely identify the release events, we could specify robust declassification as a 2-safety property on those release events: if every pair of untrusted inputs results in the same trace of confidentiality release events, the program satisfies robust declassification. Identifying confientiality release events, however, requires comparing traces with different secret inputs. A trace consists of a set of observable states, not a set of release events. Release events are identified by varying secrets; the robustness of releases is identified by varying untrusted input. Thus properly characterizing robust declassification requires 4 traces.

Both prior work [43] and our definition in Section 3.5.4 state robust declassification in terms of four traces, making it easy to convert to a 4-hyperproperty. That formulation cannot, however, be directly translated to 4-safety, as 4-safety requires a statement about trace prefixes, which cannot be invalidated by extending traces.

Instead of simply reformulating Definition 3.3 with trace prefixes, we modify it using insights gained from the definition of NMIF. In particular, instead of a strict requirement that if a relevant attack results in public-equivalent trace prefixes then other attacks must as well, we relax this requirement to apply only when the trace prefixes are trusted-equivalent. As noted in Section 3.5.6, if we syntactically prohibit endorse—the only case in which we could enforce the previous definition—this trivially reduces to that definition. Without the syntactic restriction, however, the new condition is still enforceable.

For a given attacker $\mathcal{A}$ we can define a 4-safety property with respect to $\mathcal{A}$ (let $\mathcal{U}, \mathcal{S}, \mathcal{T}, \mathcal{P}$, and $\mathcal{W}$ be as in Definition 3.5).

$$
\begin{aligned}
\mathbf{RD}_{\mathcal{A}} \triangleq \Big\{\mathbf{T} \subseteq \mathbb{T} \mid \mathbf{T} = \big\{t^{11}, t^{12}, t^{21}, t^{22}\big\} \\
\wedge\, t_1^{ij} \neq \bullet \;\wedge\; t_2^{ij} \neq \bullet \;\wedge\; t_1^{i1} = t_1^{i2} \;\wedge\; t_2^{1j} = t_2^{2j} \\
\Longrightarrow \Big(\forall\{n_{ij}\} \subseteq \mathbb{N} : (t_{n_{ij}}^{ij} \not\approx_{\mathcal{W}} \bullet \;\wedge\; t_{..n_{i1}-1}^{i1} \approx_{\mathcal{T}}^{\star} t_{..n_{i2}-1}^{i2} \\
\wedge\; t_{..n_{11}}^{11} \approx_{\mathcal{P}}^{\star} t_{..n_{21}}^{21} \;\wedge\; t_{..n_{12}}^{12} \not\approx_{\ell} \mathcal{P}^{\star} t_{..n_{22}}^{22}) \\
\Longrightarrow t_{..n_{12}}^{12} \approx_{\mathcal{W}} t_{..n_{22}}^{22}\Big)\Big\}
\end{aligned}
$$

We then define robustness against all attackers as the intersection over all attackers: $\mathbf{RD} = \bigcap_{\mathcal{A}} \mathbf{RD}_{\mathcal{A}}$.

The above definition structurally combines Definition 3.2 with the first clause of Definition 3.5 to capture both the equivalence and the relevant-input statements of the original theorem. In the nested implication, if the first two clauses hold ($t_{n_{ij}}^{ij} \not\approx_{\mathcal{W}} \bullet$ and $t_{..n_{i1}-1}^{i1} \approx_{\mathcal{T}}^{\star} t_{..n_{i2}-1}^{i2}$), then one of three things must happen when fixing the attack and varying the secret: both trace pairs are equivalent, both trace pairs are non-equivalent, or the postcondition of the implication holds ($t_{..n_{12}}^{12} \approx_{\mathcal{W}} t_{..n_{22}}^{22}$). The first two satisfy the equivalency implication in Definition 3.5 while the third is exactly a demonstration that the first input is irrelevant.

Next we note that, while this does not strictly conform to the definition of robust declassification in Definition 3.3 which cannot be stated as a hypersafety property, **RD** is equivalent to Definition 3.3 for programs that do not perform endorsement. This endorse-free condition means that the equivalence clause $t_{..n_{i1}-1}^{i1} \approx_{\mathcal{T}}^{\star} t_{..n_{i2}-1}^{i2}$ will be true whenever the trace prefixes refer to the same point in execution. In particular, they can refer to the end of execution, which gives exactly the condition specified in the theorem.

As with every other result so far, the dual construction results in a 4-safety property **TE** representing transparent endorsement. Since **RD** captures the first

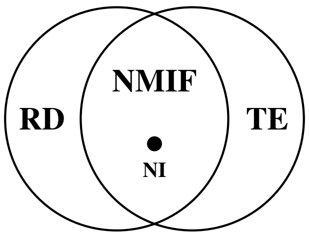


Figure 3.12: Relating 4-safety hyperproperties and noninterference.

clause of Definition 3.5, **TE** thus captures the second. We can now represent nonmalleable information flow as a 4-safety property very simply: **NMIF** = **RD** ∩ **TE**.

Figure 3.12 illustrates the relation between these hyperproperty definitions. Observe that the 2-safety hyperproperty **NI** for noninterference is contained in all three 4-safety hyperproperties. The insecure example programs of Section 3.1 are found in the left crescent, satisfying **RD** but not **NMIF**.

3.7 Implementing NMIF

We have implemented the rules for nonmalleable information flow in context of Flame [8], a Haskell library and GHC [71] plugin. Flame provides data structures and compile-time checking of type-level acts-for constraints that are checked using a custom type-checker plugin. These constraints are used as the basis for encoding NMIFC as a shallowly-embedded domain-specific language (DSL). We have demonstrated that programs enforcing nonmalleable information flow can be built using this new DSL.

3.7.1 Information-Flow Monads in Flame

The DSL works by wrapping sensitive information in an abstract data type—a monad—that includes a principal type parameter representing the confidentiality and integrity of the information.

```
class (Monad e, Labeled n) => IFC m e n where
  protect :: (pc ⊑ l) => a -> m e n pc l a

  use :: (l ⊑ l', pc ⊑ pc', l ⊑ pc', pc ⊑ pc'') =>
      m e n pc l a -> (a -> m e n pc' l' b)
                     -> m e n pc'' l' b

  runIFC :: m e n pc l a -> e (n l a)
```

Figure 3.13: Core information flow control operations in Flame.

The Flame library tracks computation on protected information as a monadic effect and provides operations that constrain such computations to enforce information security. This effect is modeled using the `IFC` type class defined in Figure 3.13. The type class `IFC` is parameterized by two additional types, `n` in the `Labeled` type class and `e` in `Monad`. Instances of the `Labeled` type class enforce noninterference on pure computation–no downgrading or effects. The `e` parameter represents an effect we want to control. For instance, many Flame libraries control effects in the `IO` monad, which is used for input, output, and mutable references.

The type `m e n pc l a` in Figure 3.13 associates a label `l` with the result of a computation of type `a`, as well as a program counter label `pc` that bounds the confidentiality and integrity of side effects for some effect `e`. Confidentiality and integrity projections are represented by type constructors `C` and `I`. The `protect` operator corresponds to monadic unit η (rule UNITM). Given any term, `protect` labels the term and lifts it into an `IFC` type where `pc ⊑ l`.

The `use` operation corresponds to a `bind` term in NMIFC. Its constraints implement the BINDM typing rule. Given a protected value of type `m e n pc l a` and a function on a value of type `a` with return type `m e n pc’ l’ b`, `use` returns the result of applying the function, provided that `l ⊑ l’` and `(pc ⊔ l) ⊑ pc’`. Finally, `runIFC` executes a protected computation, which results in a labeled value of type `(n l a)` in the desired effect `e`.

```
class IFC m e n => NMIF m e n where
  declassify :: ( (C pc) ⊑ (C l)
                , (C l') ⊑ (C l) ⊔ Δ(I (l' ⊔ pc))
                , (I l') === (I l)) =>
             m e n pc l' a -> m e n pc l a

     endorse :: ( (I pc) ⊑ (I l)
                , (I l') ⊑ (I l) ⊔ ∇(C (l' ⊔ pc))
                , (C l') === (C l)) =>
             m e n pc l' a -> m e n pc l a
```

Figure 3.14: Nonmalleable information flow control in Flame.

```
recv :: (NMIF m e n, (I p) ⊑ ∇(C p)) =>
     n p a
     -> m e n (I (p ∧ q)) (p ∧ (I q)) a
recv v = endorse $ lift v

badrecv :: (NMIF m e n, (I p) ⊑ ∇(C p)) =>
        n (p ∧ C q) a
        -> m e n (I (p ∧ q)) (p ∧ q) a
badrecv v = endorse $ lift v {-REJECTED-}
```

Figure 3.15: Receive operations in NMIF. The secure recv is accepted, but the insecure badrecv is rejected.

We provide NMIF, which extends the IFC type class with endorse and declassify operations. The constraints on these operations enforce the premises of ENDORSE and DECL, respectively.

We implemented the secure and insecure sealed-bid auction examples from Section 3.1.2 using NMIF operations, shown in Figure 3.15. As expected, the insecure badrecv is rejected by the compiler while the secure recv type checks.

3.7.2 Nonmalleable HTTP Basic Authentication

To show the utility of NMIFC, we adapt a simple existing Haskell web application [91] based on the Servant [151] framework to run in Flame. The application allows users to create, fetch, and delete shared memos. Our version uses HTTP Basic

```
authCheck :: Lbl MemoClient BasicAuthData
          -> NM IO (I MemoServer) (I MemoServer)
                  (BasicAuthResult Prin)
authCheck lauth =
 let lauth' = endorse $ lift lauth
     res = use lauth' $ \(BasicAuthData user guess) ->
           ebind user_db $ \db ->
           case Map.lookup user db of
            Nothing  -> protect Unauthorized
            Just pwd ->
              if guess == pwd then
               protect $ Authorized (Name user)
              else
               protect Unauthorized
  in declassify res
```

Figure 3.16: A nonmalleable password checker in Servant.

Authentication and Flame's security mechanisms to restrict access to authorized users. We have deployed this application online at `http://memo.flow.limited`.

Figure 3.16 contains the function `authCheck`, which checks passwords in this application using the `NM` data type, which is an instance of the `NMIF` type class. The function takes a value containing the username and password guess of the authentication attempt, labeled with the confidentiality and integrity of an unauthenticated client, `MemoClient`. This value is endorsed to have the integrity of the server, `MemoServer`. This operation is safe since it only endorses information visible to the client. Next, the username is used to look up the correct password and compare it to the client's guess. If they match, then the user is authorized. The result of this comparison is secret, so before returning the result, it must be declassified.

Enforcing any form of information flow control on authentication mechanisms like `authCheck` provides more information security guarantees than traditional approaches. Unlike other approaches, nonmalleable information flow offers strong guarantees even when a computation endorses untrusted information. This example shows it is possible to construct applications that offer these guarantees.

3.8 Related Work

Our efforts belong both within a significant body of work attempting to develop semantic security conditions that are more nuanced than noninterference, and within an overlapping body of work aiming to create expressive practical enforcement mechanisms for information flow control. Most prior work focuses on relaxing confidentiality restrictions; work permitting downgrading of integrity imposes relatively simple controls and lacks semantic security conditions that capture the concerns exemplified in Section 3.1.

Intransitive noninterference [130, 139, 142, 167] is an information flow condition that permits information to flow only between security levels (or *domains*) according to some (possibly intransitive) relation. It does not address the concerns of nonmalleability.

Decentralized information flow control (DIFC) [119] introduces the idea of mediating downgrading using access control [132]. However, the lack of robustness and transparency means downgrading can still be exploited in these systems [e.g., 65, 92, 117, 180].

Robust declassification and qualified robustness have been explored in DIFC systems as a way to constrain the adversary's influence on declassification [9, 11, 12, 44, 120, 177, 178]. While transparent endorsement can be viewed as an integrity counterpart to robust declassification, this idea is not present in prior work.

Sabelfeld and Sands provide a clarifying taxonomy for much prior work on declassification [146], introducing various dimensions along which declassification mechanisms operate. They categorize robust declassification as lying on the "who" dimension. However, they do not explicitly consider endorsement mechanisms. Regardless of the taxonomic category, transparent endorsement and nonmalleable information flow also seem to lie on the same dimension as robust declassification, since they take into account influences on the information that is downgraded.

Label algebras [113] provide an abstract characterization of several DIFC systems. However, they do not address the restrictions on downgrading imposed by nonmalleable information flow.

The Aura language [87] uses information flow policies to constrain authorization and endorsement. However, it does not address the malleability of endorsement. Rx [161] represents information flow control policies in terms of dynamic *roles* [147]. Adding new principals to these roles corresponds to declassification and endorsement since new flows may occur. Rx constrains updates to roles similarly to previous type systems that enforce robust declassification and qualified robustness but does not prevent opaque endorsements.

Relational Hoare Type Theory (RHTT) [122] offers a powerful and precise way to specify security conditions that are 2-hyperproperties, such as noninterference. Cartesian Hoare logic (CHL) [157] extends standard Hoare logic to reason about *k*-safety properties of relational traces (the input/output pairs of a program). Since nonmalleable information flow, robust declassification, and transparent endorsement are all 4-safety properties that cannot be fully expressed with relational traces, neither RHTT nor CHL can characterize them properly.

Haskell's type system has been attractive target for embedding information flow checking [28, 97, 159]. Much prior work has focused on dynamic information flow control. LIO [159] requires computation on protected information to occur in the LIO monad, which tracks the confidentiality and integrity of information accessed ("unlabeled") by the computation. HLIO [28] explores hybrid static and dynamic enforcement. Flame enforces information flow control statically, and the NMIF type class enforces nonmalleable IFC statically as well. The static component

$n \in \mathcal{N}$ (atomic principals)
$x \in \mathcal{V}$ (variable names)

$$
\begin{array}{rcl}
p, \ell, pc & ::= & n \mid \top \mid \bot \mid p^{\pi} \mid p \wedge p \mid p \vee p \mid p \sqcup p \mid p \sqcap p \\[1ex]
\tau & ::= & \textsf{unit} \mid X \mid (\tau + \tau) \mid (\tau \times \tau) \\
& \mid & \tau \xrightarrow{pc} \tau \mid \forall X[pc].\, \tau \mid \ell \textsf{ says } \tau \\[1ex]
v & ::= & () \mid \textsf{in}_i\, v \mid \langle v, v \rangle \mid (\overline{\eta}_\ell\, v) \\
& \mid & \lambda\, x\!:\!\tau[pc].\, e \mid \Lambda X[pc].\, e \\[1ex]
e & ::= & x \mid v \mid e\; e \mid e\; \tau \mid \langle e, e \rangle \mid (\eta_\ell\, e) \\
& \mid & \textsf{proj}_i\, e \mid \textsf{in}_i\, e \mid \textsf{bind } x = e \textsf{ in } e \\
& \mid & \textsf{match } e \textsf{ with} \mid \textsf{in}_1(x).\, e \mid \textsf{in}_2(x).\, e \textsf{ end} \\
& \mid & \textsf{decl } e \textsf{ to } \ell \mid \textsf{endorse } e \textsf{ to } \ell
\end{array}
$$

Figure 3.17: Full NMIFC syntax.

of HLIO enforces solely via the Haskell type system (and existing general-purpose extensions), but Flame—and by extension, NMIF—uses custom constraints based on the FLAM algebra which are processed by a GHC type checker plugin. Extending the type checker to reason about FLAM constraints significantly improves programmability over pure-Haskell approaches like HLIO.

3.9 Full NMIFC

We present the full syntax, semantics, and typing rules for NMIFC in Figures 3.17, 3.18, and 3.20, respectively. This is a straightforward extension of the core language presented in Section 3.4. We note that polymorphic terms specify a pc just as λ terms. This is because they contain arbitrary expressions which could produce arbitrary effects, so we must constrain the context that can execute those effects.

Figure 3.21 presents the full set of derivation rules for the acts-for (delegation) relation $p \Rightarrow q$.

$\boxed{e \longrightarrow e'}$

[E-APP] $$\frac{}{(\lambda\, x\!:\!\tau[pc].\, e)\ v \longrightarrow e[x \mapsto v]}$$

[E-TAPP] $$\frac{}{(\Lambda X[pc].\, e)\ \tau \longrightarrow e[X \mapsto \tau]}$$

[E-UNPAIR] $$\frac{}{\mathsf{proj}_i\ \langle v_1, v_2\rangle \longrightarrow v_i}$$

[E-MATCH] $$\frac{}{(\mathsf{match}\ (\mathsf{in}_i\ v)\ \mathsf{with}\ |\ \mathsf{in}_1(x).\, e_1\ |\ \mathsf{in}_2(x).\, e_2\ \mathsf{end}) \longrightarrow e_i[x \mapsto v]}$$

[E-BINDM] $$\frac{}{\mathsf{bind}\ x = (\overline{\eta}_\ell\ v)\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]}$$

$\boxed{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t'\rangle}$

[E-STEP] $$\frac{e \longrightarrow e'}{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t; \bullet\rangle}$$

[E-UNITM] $$\frac{}{\langle(\eta_\ell\ v),\, t\rangle \twoheadrightarrow \langle(\overline{\eta}_\ell\ v),\, t; (\overline{\eta}_\ell\ v)\rangle}$$

[E-DECL] $$\frac{}{\langle \mathsf{decl}\ (\overline{\eta}_{\ell'}\ v)\ \mathsf{to}\ \ell,\, t\rangle \twoheadrightarrow \Big\langle(\overline{\eta}_\ell\ v),\, t; (\downarrow^{\rightarrow}_{\ell'},\, \overline{\eta}_\ell\ v)\Big\rangle}$$

[E-ENDORSE] $$\frac{}{\langle \mathsf{endorse}\ (\overline{\eta}_{\ell'}\ v)\ \mathsf{to}\ \ell,\, t\rangle \twoheadrightarrow \Big\langle(\overline{\eta}_\ell\ v),\, t; (\downarrow^{\leftarrow}_{\ell'},\, \overline{\eta}_\ell\ v)\Big\rangle}$$

[E-EVAL] $$\frac{\langle e,\, t\rangle \twoheadrightarrow \langle e',\, t'\rangle}{\langle E[e],\, t\rangle \twoheadrightarrow \langle E[e'],\, t'\rangle}$$

Evaluation context

$$\begin{array}{rcl} E & ::= & [\cdot]\ |\ E\ e\ |\ v\ E\ |\ E\ \tau\ |\ \langle E, e\rangle\ |\ \langle v, E\rangle\ |\ (\eta_\ell\ E) \\ & | & \mathsf{proj}_i\ E\ |\ \mathsf{in}_i\ E\ |\ \mathsf{bind}\ x = E\ \mathsf{in}\ e \\ & | & \mathsf{match}\ E\ \mathsf{with}\ |\ \mathsf{in}_1(x).\, e\ |\ \mathsf{in}_2(x).\, e\ \mathsf{end} \\ & | & \mathsf{decl}\ E\ \mathsf{to}\ \ell\ |\ \mathsf{endorse}\ E\ \mathsf{to}\ \ell \end{array}$$

Figure 3.18: Full NMIFC operational semantics.

$\boxed{\ell \triangleleft \tau}$

$$[\text{P-U}\textsc{nit}]\ \ell \triangleleft \mathsf{unit} \qquad [\text{P-L}\textsc{bl}]\ \frac{\ell' \sqsubseteq \ell}{\ell' \triangleleft \ell \ \mathsf{says}\ \tau} \qquad [\text{P-P}\textsc{air}]\ \frac{\ell \triangleleft \tau_1 \qquad \ell \triangleleft \tau_2}{\ell \triangleleft (\tau_1 \times \tau_2)}$$

$\boxed{\tau \triangleleft \mathcal{H}}$

$$[\text{P-S}\textsc{et}]\ \frac{H \in \mathcal{H} \qquad H \triangleleft \tau}{\vdash \tau\ \mathsf{prot}\ \mathcal{H}}\ \mathcal{H} \text{ is upward closed}$$

Figure 3.19: Type protection levels.

3.9.1 Label Tracking with Brackets

In order to simply proofs of hyperproperties requiring 2 and 4 traces, we introduce a new bracket syntax to track secret and untrusted data. These brackets are inspired by those used by Pottier and Simonet [133] to prove their FlowCaml type system enforced noninterference. Their brackets served two purposes simultaneously. First they allow a single execution of a bracketed program to faithfully model two executions of a non-bracketed program. Second, the brackets track secret/untrusted information through execution of the program, thereby making it easy to verify that it did not interfere with public/trusted information simply by proving that brackets could not be syntactically present in such values. Since noninterference only requires examining pairs of traces, these purposes complement each other well; if the two executions vary only on high inputs, then low outputs cannot contain brackets.

While this technique is very effective to prove noninterference, nonmalleable information flow provides security guarantees even in the presence of both declassification and endorsement. As a result, we need to track secret/untrusted information even through downgrading events that can cause traces to differ arbitrarily. To accomplish this goal, we use brackets that serve only the second purpose: they

$\boxed{\Gamma; pc \vdash e : \tau}$

$$[\textsc{Var}]\ \frac{\Gamma(x) = \tau}{\Gamma; pc \vdash x : \tau} \qquad [\textsc{Unit}]\ \frac{}{\Gamma; pc \vdash () : \mathsf{unit}}$$

$$[\textsc{Lam}]\ \frac{\Gamma, x{:}\tau_1; pc' \vdash e : \tau_2}{\Gamma; pc \vdash \lambda\, x{:}\tau_1[pc'].\, e : \tau_1 \xrightarrow{pc'} \tau_2} \qquad [\textsc{App}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e_1 : \tau' \xrightarrow{pc'} \tau \\ \Gamma; pc \vdash e_2 : \tau' \qquad pc \sqsubseteq pc'\end{array}}{\Gamma; pc \vdash e_1\ e_2 : \tau}$$

$$[\textsc{TLam}]\ \frac{\Gamma, X; pc' \vdash e : \tau}{\Gamma; pc \vdash \Lambda X[pc'].\, e : \forall X[pc'].\, \tau}$$

$$[\textsc{TApp}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \forall X[pc'].\, \tau \\ pc \sqsubseteq pc'\end{array}}{\Gamma; pc \vdash (e\ \tau') : \tau[X \mapsto \tau']}\ \tau' \text{ is well-formed in } \Gamma$$

$$[\textsc{Pair}]\ \frac{\Gamma; pc \vdash e_1 : \tau_1 \qquad \Gamma; pc \vdash e_2 : \tau_2}{\Gamma; pc \vdash \langle e_1, e_2 \rangle : (\tau_1 \times \tau_2)} \qquad [\textsc{UnPair}]\ \frac{\Gamma; pc \vdash e : (\tau_1 \times \tau_2)}{\Gamma; pc \vdash \mathsf{proj}_i\ e : \tau_i}$$

$$[\textsc{Inj}]\ \frac{\Gamma; pc \vdash e : \tau_i}{\Gamma; pc \vdash \mathsf{in}_i\ e : (\tau_1 + \tau_2)} \qquad [\textsc{Match}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e : (\tau_1 + \tau_2) \qquad pc \triangleleft \tau \\ \Gamma, x{:}\tau_1; pc \vdash e_1 : \tau \qquad \Gamma, x{:}\tau_2; pc \vdash e_2 : \tau\end{array}}{\Gamma; pc \vdash \mathsf{match}\ e\ \mathsf{with}\ |\ \mathsf{in}_1(x).\, e_1\ |\ \mathsf{in}_2(x).\, e_2\ \mathsf{end} : \tau}$$

$$[\textsc{UnitM}]\ \frac{\Gamma; pc \vdash e : \tau \qquad pc \sqsubseteq \ell}{\Gamma; pc \vdash (\eta_\ell\ e) : \ell\ \mathsf{says}\ \tau} \qquad [\textsc{VUnitM}]\ \frac{\Gamma; pc \vdash v : \tau}{\Gamma; pc \vdash (\overline{\eta}_\ell\ v) : \ell\ \mathsf{says}\ \tau}$$

$$[\textsc{BindM}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \ell\ \mathsf{says}\ \tau' \qquad \ell \triangleleft \tau \\ \Gamma, x{:}\tau'; pc \sqcup \ell \vdash e' : \tau\end{array}}{\Gamma; pc \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \tau}$$

$$[\textsc{Decl}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \ell'\ \mathsf{says}\ \tau \qquad pc \sqsubseteq \ell \\ \ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow}) \qquad \ell'^{\leftarrow} = \ell^{\leftarrow}\end{array}}{\Gamma; pc \vdash \mathsf{decl}\ e\ \mathsf{to}\ \ell : \ell\ \mathsf{says}\ \tau}$$

$$[\textsc{Endorse}]\ \frac{\begin{array}{c}\Gamma; pc \vdash e : \ell'\ \mathsf{says}\ \tau \qquad pc \sqsubseteq \ell \\ \ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow}) \qquad \ell'^{\rightarrow} = \ell^{\rightarrow}\end{array}}{\Gamma; pc \vdash \mathsf{endorse}\ e\ \mathsf{to}\ \ell : \ell\ \mathsf{says}\ \tau}$$

Figure 3.20: Typing rules for full NMIFC language.

$\boxed{p \Rightarrow q}$

$$\text{[Bot]}\ \frac{}{p \Rightarrow \bot} \qquad \text{[Top]}\ \frac{}{\top \Rightarrow p} \qquad \text{[Refl]}\ \frac{}{p \Rightarrow p} \qquad \text{[ProjR]}\ \frac{}{p \Rightarrow p^{\pi}}$$

$$\text{[Proj]}\ \frac{p \Rightarrow q}{p^{\pi} \Rightarrow q^{\pi}} \qquad \text{[ConjL]}\ \frac{\begin{array}{c} p_i \Rightarrow q \\ i \in \{1,2\} \end{array}}{p_1 \wedge p_2 \Rightarrow q} \qquad \text{[ConjR]}\ \frac{\begin{array}{c} p \Rightarrow q_1 \\ p \Rightarrow q_2 \end{array}}{p \Rightarrow q_1 \wedge q_2}$$

$$\text{[DisL]}\ \frac{\begin{array}{c} p_1 \Rightarrow q \\ p_2 \Rightarrow q \end{array}}{p_1 \vee p_2 \Rightarrow q} \qquad \text{[DisR]}\ \frac{\begin{array}{c} p \Rightarrow q_i \\ i \in \{1,2\} \end{array}}{p \Rightarrow q_1 \vee q_2} \qquad \text{[Trans]}\ \frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r}$$

Figure 3.21: Principal lattice rules

track restricted information but not multiple executions.

As in previous formalizations, NMIFC's brackets are defined with respect to a notion of "high" labels, in this case a high set. The high set restricts the type of the expression inside the bracket as well as the *pc* at which it must type, thereby restricting the effects it can create. For the more complex theorems we must track data with multiple different high labels within the same program execution, so we parameterize the brackets themselves with the high set. We present the extended syntax, semantics, and typing rules in Figure 3.22.

3.10 Attacker Properties

Recall that we defined an attacker as a set of principals $\mathcal{A} = \{\ell \in \mathcal{L} \mid n_1 \wedge \cdots \wedge n_k \Rightarrow \ell\}$ for some non-empty finite set of atomic principals $\{n_1, \ldots, n_k\} \subseteq \mathcal{N}$. The formal results of this work rely only on a somewhat more general definition of an attacker.

Definition 3.6 (Attacker)**.** An attacker $\mathcal{A} \subseteq \mathcal{L}$ is any subset of $\mathcal{L}$ with the following properties:

1. $\mathcal{A}$ is upward-closed in the trust ordering: for all $a \in \mathcal{A}$ and $b \in \mathcal{L}$, if $a \Rightarrow b$,

Syntax extensions

$$\begin{array}{rcl} v & ::= & \cdots \mid (\!| v |\!)_{\mathcal{H}} \\ e & ::= & \cdots \mid (\!| e |\!)_{\mathcal{H}} \end{array}$$

New contexts

$$\begin{array}{rcl} E & ::= & \cdots \mid (\!| E |\!)_{\mathcal{H}} \\ B & ::= & \mathsf{proj}_i\ [\cdot] \mid \mathsf{bind}\ x = [\cdot]\ \mathsf{in}\ e \end{array}$$

Evaluation extensions

[B-EXPAND] $$B[(\!| v |\!)_{\mathcal{H}}] \longrightarrow (\!| B[v] |\!)_{\mathcal{H}}$$

[B-DECLL] $$\frac{\ell \notin \mathcal{H}}{\mathsf{decl}\ (\!| v |\!)_{\mathcal{H}}\ \mathsf{to}\ \ell \longrightarrow \mathsf{decl}\ v\ \mathsf{to}\ \ell}$$

[B-DECLH] $$\frac{\ell \in \mathcal{H}}{\mathsf{decl}\ (\!| v |\!)_{\mathcal{H}}\ \mathsf{to}\ \ell \longrightarrow (\!| \mathsf{decl}\ v\ \mathsf{to}\ \ell |\!)_{\mathcal{H}}}$$

[B-ENDORSEL] $$\frac{\ell \notin \mathcal{H}}{\mathsf{endorse}\ (\!| v |\!)_{\mathcal{H}}\ \mathsf{to}\ \ell \longrightarrow \mathsf{endorse}\ v\ \mathsf{to}\ \ell}$$

[B-ENDORSEH] $$\frac{\ell \in \mathcal{H}}{\mathsf{endorse}\ (\!| v |\!)_{\mathcal{H}}\ \mathsf{to}\ \ell \longrightarrow (\!| \mathsf{endorse}\ v\ \mathsf{to}\ \ell |\!)_{\mathcal{H}}}$$

Typing extensions

[BRACKET] $$\frac{\Gamma; pc' \vdash e : \tau \qquad pc \sqsubseteq pc' \qquad pc' \in \mathcal{H} \qquad \vdash \tau\ \mathsf{prot}\ \mathcal{H}}{\Gamma; pc \vdash (\!| e |\!)_{\mathcal{H}} : \tau}\ \mathcal{H}\text{ is upward closed}$$

Bracket projection

$$\lfloor e \rfloor = \begin{cases} \lfloor e' \rfloor & \text{if } e = (\!| e' |\!)_{\mathcal{H}} \\ \text{recursively project all sub-expressions otherwise} \end{cases}$$

Figure 3.22: NMIFC language extensions.

then $b \in \mathcal{A}$.

2. $\mathcal{A}$ is a sublattice of $\mathcal{L}$: for all $a, b \in \mathcal{A}$, $a \vee b \in \mathcal{A}$ and $a \wedge b \in \mathcal{A}$.
3. The complement $\overline{\mathcal{A}} = \mathcal{L} \setminus \mathcal{A}$ is also a sublattice of $\mathcal{L}$.
4. For all $a \in \mathcal{A}$, $\nabla(a^{\rightarrow}) \wedge \Delta(a^{\leftarrow}) \in \mathcal{A}$.

Note that Conditions 2 and 3 imply that $\mathcal{A}^{\rightarrow}$ and $\mathcal{A}^{\leftarrow}$ and their complements are also sublattices of $\mathcal{L}$.

3.11 Generalization

Definition 3.5 (and correspondingly Theorem 3.6) might appear relatively narrow; they only speak directly to programs with a single untrusted value and a single secret value. However, because the language has first-class functions and pair types, the theorem as stated is equivalent to one that allows insertion of secret and untrusted code into multiple points in the program, as long as that code types in an appropriately restrictive *pc*.

To define this formally, we first need a means to allow for insertion of arbitrary code. We follow previous work [120] by extending the language to include *holes*. A program expression may contain an ordered set of holes. These holes may be replaced with arbitrary expressions, under restrictions requiring that the holes be treated as sufficiently secret or untrusted. Specifically, the type system is extended with the following rule:

$$[\textsc{Hole}]\ \frac{pc \in \mathcal{H} \qquad \vdash \tau\ \mathsf{prot}\ \mathcal{H}}{\Gamma; pc \vdash [\bullet]_{\mathcal{H}} : \tau}\ \mathcal{H}\text{ is a high set}$$

Using this definition, we can state NMIF in a more traditional form.

Definition 3.7 (General NMIF)**.** We say that a program $e[\vec{\bullet}]_{\mathcal{H}}$ enforces *general NMIF* if the following holds for all attackers $\mathcal{A}$ inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$, $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$ and $\mathcal{W} = \mathcal{T} \cap \mathcal{S}$. If $\mathcal{H} \subseteq \mathcal{U}$, then for all values v_1, v_2 and all attacks $\vec{a}_1$ and $\vec{a}_2$, let

$$\langle e[\vec{a}_i]_{\mathcal{H}}[\vec{x} \mapsto \vec{v}_i],\, \vec{v}_i\rangle \longrightarrow\!\!\!\!\rightarrow^* \left\langle v_{ij},\, t^{ij}\right\rangle.$$

For all indices n_{ij} such that $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$

1. If $t^{i1}_{..n_{i1}-1} \approx^{\star}_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ for $i = 1, 2$, then

$$\left(\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{P}} t^{21}_{..n_{21}}\right) \implies t^{12}_{..n_{12}} \approx^{\star}_{\mathcal{P}} t^{22}_{..n_{22}}.$$

2. Similarly, if $t^{1j}_{..n_{1j}-1} \approx^{\star}_{\mathcal{P}} t^{2j}_{..n_{2j}-1}$ for $j = 1, 2$, then

$$\left(\mathrm{rel}^{\rightarrow}_{\mathcal{A},e}(v_1) \text{ and } t^{11}_{..n_{11}} \approx^{\star}_{\mathcal{T}} t^{12}_{..n_{12}}\right) \implies t^{21}_{..n_{21}} \approx^{\star}_{\mathcal{T}} t^{22}_{..n_{22}}.$$

For NMIFC, this definition is equivalent to Definition 3.5. We prove this fact to prove the following theorem.

Theorem 3.7 (General NMIF)**.** *Given a program $e[\vec{\bullet}]_{\mathcal{H}}$ such that $\Gamma, \vec{x}{:}\vec{\tau}; pc \vdash e[\vec{\bullet}]_{\mathcal{H}} : \tau'$, then $e[\vec{\bullet}]_{\mathcal{H}}$ enforces general NMIF.*

Proof. We prove this by reducing Definition 3.7 to Definition 3.5 in two steps. We assume that no two variables in the original expression $e[\vec{\bullet}]_{\mathcal{H}}$ have the same name as this can be enforced by α-renaming.

The first step handles expressions that only substitute values (and have no holes), but allow any number of both secret and untrusted values. An expression of the form in this corollary is easily rewritten as such a substitution as follows. For each hole $[\bullet]_{\mathcal{H}}$, we note that $\Gamma'; pc' \vdash [\bullet]_{\mathcal{H}} : \tau''$ where $\Gamma, \vec{x}{:}\vec{\tau} \subseteq \Gamma'$ and $pc' \in \mathcal{H}$. We replace the hole with a function application inside a bind. Specifically, the hole becomes

$$\mathsf{bind}\ y' = y\ \mathsf{in}\ (y'\ z_1\ \cdots\ z_k)$$

where y and y' are fresh variables and the z_is are every variable in $\Gamma' \setminus \Gamma$ (including every element of $\vec{x}$). Let

$$\tau_y = pc' \ \mathsf{says} \ \left(\tau_{z_1} \xrightarrow{pc'} \cdots \xrightarrow{pc'} \tau_{z_k} \xrightarrow{pc'} \tau''\right)$$

and include $y : \tau_y$ as the type of an untrusted value to substitute in.

Instead of inserting the expression a into that hole, we substitute in for y the value

$$w = \overline{\eta}_{pc'} \ (\lambda z_1 : \tau_{z_1}[pc']. \ \cdots \ \lambda z_k : \tau_{z_k}[pc']. \, a).$$

By HOLE we know that $pc' \in \mathcal{H}$ and $\vdash \tau'' \ \mathsf{prot} \ \mathcal{H}$, so the type has the proper protection, and by construction $\Gamma; pc \vdash w : \tau_y$. Moreover, while it has an extra value at the beginning of the trace (the function), the rest of the traces are necessarily the same.

As a second step, we reduce the rest of the way to the expressions used in Definition 3.5. To get from our intermediate step to these single-value expressions, if we wish to substitute k_s secret values and k_u untrusted values, we instead substitute a single list of k_s secret values and a single list of k_u untrusted values. These lists are constructed in the usual way out of pairs, meaning the protection relations continue to hold as required. Finally, whenever a variable is referenced in the unsubstituted expression, we instead select the appropriate element out of the substituted list using nested projections. □

We also note that the same result holds if we allow for insertion of secret code and untrusted values, as the argument is exactly dual. Such a situation, however, makes less sense, so we do not present it explicitly.

3.12 Proofs

We now prove a variety of properties about NMIFC.

3.12.1 Language Results

We begin with the core results about the language itself.

Lemma 3.1 (Values)**.** *For any value v such that $\Gamma; pc \vdash v : \tau$, $\Gamma; pc' \vdash v : \tau$ for any pc'.*

Proof. This follows by induction on the typing derivation for values. □

Lemma 3.2 (Substitution)**.** *If $\Gamma, x{:}\tau'; pc \vdash e : \tau$ and $\Gamma; pc \vdash v : \tau'$, then $\Gamma; pc \vdash e[x \mapsto v] : \tau$.*

Proof. By induction on the derivation of $\Gamma, x{:}\tau'; pc \vdash e : \tau$ using Lemma 3.1. □

Lemma 3.3 (*pc* reduction)**.** *If $\Gamma; pc \vdash e : \tau$ and $pc' \sqsubseteq pc$, then $\Gamma; pc' \vdash e : \tau$.*

Proof. By induction on the derivation of $\Gamma; pc \vdash e : \tau$. □

Theorem 3.8 (Subject reduction)**.** *If $\Gamma; pc \vdash e : \tau$ and $\langle e,\ t\rangle \longrightarrow\!\!\!\!\rightarrow \langle e',\ t'\rangle$ then $\Gamma; pc \vdash e' : \tau$.*

Proof. This proof follows by an inductive case analysis on the operational semantics in Figures 3.18 and 3.22. There are a few interesting cases.

Case B-EXPAND**:** This case handles a context (B), we will do a sub-case analysis on each such expression type.

- $e = (\mathsf{proj}_i\ (\!|v|\!)_{\mathcal{H}})$: UNPAIR allows the expression to type-check in any pc in which its argument type-checks. Since BRACKET says $\Gamma; pc' \vdash v : (\tau_1 \times \tau_2)$ for some $pc' \in \mathcal{H}$, we get that $\Gamma; pc' \vdash \mathsf{proj}_i\ v : \tau_i$. Moreover BRACKET and P-SET also require $H \triangleleft (\tau_1 \times \tau_2)$ which, by P-PAIR can only happen if $H \triangleleft \tau_i$ for $i = 1, 2$. Together these satisfy the requirements for BRACKET and give us $\Gamma; pc' \vdash (\!|\mathsf{proj}_i\ v|\!)_{\mathcal{H}} : \tau_i$.
- $e = (\mathsf{bind}\ x = (\!|v|\!)_{\mathcal{H}}\ \mathsf{in}\ e')$: By BINDM, $\Gamma; pc \vdash (\!|v|\!)_{\mathcal{H}} : \ell\ \mathsf{says}\ \tau'$ and by BRACKET and inversion on the protection rules, $H \sqsubseteq \ell$ for some $H \in \mathcal{H}$ and thus $\ell \in \mathcal{H}$. Let $pc' = pc'' \sqcap (pc \sqcup \ell)$ where pc'' is the higher pc used in BRACKET.

Since $\mathcal{H}$ is upward closed and $pc'', \ell \in \mathcal{H}$, $(pc \sqcup \ell) \in \mathcal{H}$ and thus $pc' \in \mathcal{H}$. Moreover, $pc' \sqcup \ell = pc \sqcup \ell$, so by BINDM $\Gamma, x{:}\tau'; pc' \sqcup \ell \vdash e' : \tau$ and by Lemma 3.1 $\Gamma; pc' \vdash v : \ell \textsf{ says } \tau'$. This means that $\Gamma; pc' \vdash \textsf{bind } x = v \textsf{ in } e' : \tau$. Also by BINDM we have $\ell \triangleleft \tau$ so since $\ell \in \mathcal{H}$ we have now satisfied the conditions on BRACKET and $\Gamma; pc \vdash (\!|\textsf{bind } x = v \textsf{ in } e'|\!)_{\mathcal{H}} : \tau$.

Case B-DECLH**:** By DECL $\Gamma; pc \vdash (\!|v|\!)_{\mathcal{H}} : \ell' \textsf{ says } \tau'$ and by inspection on the protection rules and BRACKET, it must be the case that $H \sqsubseteq \ell'$ for some $H \in \mathcal{H}$ and thus $\ell' \in \mathcal{H}$.

By assumption $\ell \in \mathcal{H}$, so the final protection requirement of BRACKET on the stepped expression is satisfied. We now claim that if $pc' = pc \sqcup \ell$ then $\Gamma; pc' \vdash \textsf{decl } v \textsf{ to } \ell : \tau$. DECL gives us that $pc \sqsubseteq \ell$, so clearly $pc' = pc \sqcup \ell \sqsubseteq \ell$.

DECL also gives us $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})$. Since $\ell'^{\leftarrow} = \ell^{\leftarrow}$, we note that

$$(\ell' \sqcup pc)^{\leftarrow} = (\ell \sqcup pc)^{\leftarrow} = pc'^{\leftarrow}.$$

Therefore $(\ell' \sqcup pc')^{\leftarrow} = (\ell' \sqcup pc)^{\leftarrow}$ and the premise still holds.

Since the other premises DECL are trivially still satisfied in a higher pc (using Lemma 3.1 for v to type), we see that $\Gamma; pc' \vdash \textsf{decl } v \textsf{ to } \ell : \tau$. Thus since $pc' \in \mathcal{H}$ and $\ell \in \mathcal{H}$, we get $\Gamma; pc \vdash (\!|\textsf{decl } v \textsf{ to } \ell|\!)_{\mathcal{H}} : \tau$.

Case B-ENDORSEH**:** We omit the details of this case as they are identical to the previous case, but with projection arrows reversed and Δ replaced by ∇.

All other cases follow trivially from inspection on the typing derivations in Figure 3.8 and applications of Lemmas 3.2 and 3.3. Rule E-EVAL also requires an inductive application. □

Lemma 3.4 (Bracket Soundness)**.** *If* $\langle e, \varepsilon\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle e', t\rangle$ *then* $\langle \lfloor e \rfloor, \varepsilon\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle \lfloor e' \rfloor, \lfloor t \rfloor\rangle$.

Proof. This result follows by inspection on the operational semantics in Figures 3.18 and 3.22. Note that every step the non-projected expression takes is mirrored exactly by a step the projected value takes except applications of B-EXPAND, B-DECLL, B-DECLH, B-ENDORSEL, and B-ENDORSEH. Those applications are simply dropped and since they emit no values to the trace, the traces remain the same. □

Lemma 3.5 (Bracket Completeness). *If* $\langle \lfloor e \rfloor, \varepsilon \rangle \longrightarrow\!\!\!\!\!\rightarrow^* \langle e', t \rangle$ *and* $\Gamma; pc \vdash e : \tau$, *then there is some* e'', t' *such that* $\langle e, \varepsilon \rangle \longrightarrow\!\!\!\!\!\rightarrow^* \langle e'', t' \rangle$.

Proof. Assume $\langle \lfloor e \rfloor, \varepsilon \rangle \longrightarrow\!\!\!\!\!\rightarrow^* \langle e', t \rangle$ and consider the operational semantics in Figures 3.18 and 3.22. We consider a single step in the projected expression which gives us two cases.

If the step happens either entirely within a bracket or entirely outside brackets, then the same step must be possible in the original expression. If the step in the original expression is not possible because of brackets, then there must be brackets around a term other than an arbitrary expression or value in the semantic rule employed. Since the expression type-checks and by Theorem 3.8, each intermediate expression also type-checks, the protection clause on BRACKET and inversion on the protection rules in Figure 3.7 give us four possible options for the semantic rule employed: E-UNPAIR, E-BINDM, E-DECL, and E-ENDORSE. For the first two, we can first step using B-EXPAND one or more times before applying the original rule. For the second two, we can apply B-DECLL, B-DECLH, B-ENDORSEL, or B-ENDORSEH one or more times again before applying the original rule.

In all cases we see that if the projected term steps and equivalent step is possible in the original expression, though possibly after applying one or more other steps first. This proves the desired result. □

3.12.2 Security Results

We now prove the various security results stated throughout the paper.

Lemma 3.6 (Release on downgrade)**.** *Let $\mathcal{H}$ be a high set and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given a program e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ with $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, let*

$$\langle e[x \mapsto (\!| v_i |\!)_{\mathcal{H}}],\ v_i \rangle \longrightarrow\!\!\!\!\!\rightarrow^* \langle v_i',\ t_i \rangle .$$

If n_1 and n_2 are such that $t^i_{n_i} \not\approx_{\mathcal{W}} \bullet$ and $t^1_{..n_1-1} \approx^{\star}_{\mathcal{W}} t^2_{..n_2-1}$, then either $t^i_{n_i} = (\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w_i)$ with $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$ for both $i = 1, 2$, or $t^1_{n_1} \approx_{\mathcal{W}} t^2_{n_2}$.

Proof. We refer to elements of the form $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w)$ with $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$ as *relevant downgrade* elements. This is a proof by induction on the number of relevant downgrade elements in $t^1_{..n_1-1}$ and $t^2_{..n_2-1}$. Note that while the prefixes can contain any number of relevant downgrade elements, the prefixes are $\mathcal{W}$-equivalent, so the downgraded values must also be $\mathcal{W}$-equivalent. In particular, there must be the same number in each trace prefix.

As the base case, assume there are no such values in the trace prefixes. We first claim that there can be no application of B-DECLL or B-ENDORSEL prior to the current step.

By the typing rules DECL and ENDORSE, that would require the value inside the bracket to have type ℓ' says τ for some type τ. The protection clause on BRACKET and the protection rules in Figure 3.19 would thus require there to be some $H \in \mathcal{H}$ such that $H \sqsubseteq \ell'$, which in turn means $\ell' \in \mathcal{H}$. However, application of B-DECLL or B-ENDORSEL requires $\ell \notin \mathcal{H}$. Therefore if either rule is applied, the next value emitted to the trace must be $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w)$ where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$. While $t^i_{n_i}$ may be of that form, we assumed that no prior values are.

Next we claim that any differences between the two prefixes must be emitted from within brackets. The initial expression in each case differs only in the substituted value, which is inside brackets. If we syntactically examine the expression at every step of evaluation, we notice that no semantic rules allow anything inside brackets to affect anything outside brackets except through the B-DECLL and B-ENDORSEL rules, which remove brackets. Since those rules are never applied, all differences must be contained within brackets. This means that all differences within the prefixes must have been emitted from inside brackets.

Finally, we claim that if a trace element c is emitted from within a bracket, then $c \approx_{\mathcal{W}} \bullet$. The only three rules that emit trace elements are E-UNITM, E-DECL, and E-ENDORSE. By Theorem 3.8, the expression stepping must type check, and each of the corresponding typing rules (UNITM, DECL, and ENDORSE) contain the condition $pc \sqsubseteq \ell$. By BRACKET, if the expression is inside a bracket then $pc \in \mathcal{H}$ and since $\mathcal{H}$ is upward closed, this means $\ell \in \mathcal{H}$. Thus by EQ-UNITM or EQ-DOWN, $c \approx_{\mathcal{W}} \bullet$.

Coupled with the above logic about applications of B-DECLL and B-ENDORSEL and the fact that $t^i_{n_i} \not\approx_{\mathcal{W}} \bullet$, either $t^1_{n_1} \approx_{\mathcal{W}} t^2_{n_2}$, or both are the result of downgrades that resulted in applications of either B-DECLL or B-ENDORSEL. In the latter case both are of the form $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_{\ell}\ w)$ where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, as desired.

Now we assume that there is at least one such expression in the trace prefixes, but those prefixes are still $\mathcal{W}$-equivalent. Take the first such trace element. This element must appear in both trace prefixes. Let e_1 and e_2 be the top-level expressions that stepped in each trace to emit the downgrade element. The two expressions can differ inside brackets, and can differ inside the downgraded value (that was previously inside brackets). Because the downgraded values are $\mathcal{W}$-equivalent, any such differences must be contained inside terms of the form $(\overline{\eta}_{\ell}\ w)$ where $\ell \in \mathcal{H}$. We can replace any such terms by $(\!| \overline{\eta}_{\ell}\ w |\!)_{\mathcal{H}}$ in the expression and it will still type-check since $\ell \in \mathcal{H}$ and w type-checks in any pc. This results in a new pair of expressions that are $\mathcal{W}$-equivalent except inside brackets. By Lemmas 3.4 and 3.5 these expression generate traces that are equivalent to the original up to brackets and bullets. By construction, these new expressions generate trace prefixes with one less $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w)$ element than the original expressions, so by the inductive hypothesis, the result holds.

Since the same brackets were necessarily added in both expressions, the new expressions will thus generate equivalent traces if and only if the old expressions did. Since the trace prefixes prior to this modification were equivalent, if the new expressions generate equivalent traces, the original expressions must also generate equivalent traces. Thus the result holds for the original traces as well. $\square$

Theorem 3.1 (Noninterference of non-downgrading programs)**.** *Let $\mathcal{H}$ be a high set and let $\mathcal{W} = \mathcal{L} \backslash \mathcal{H}$. Given an expression e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if*

$$\langle e[x \mapsto v_i],\ v_i \rangle \longrightarrow\!\!\!\!\!\rightarrow^{*} \left\langle v_i',\ t^i \right\rangle$$

then either there is some $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w) \in t^i$ where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, or $t^1 \approx^{\star}_{\mathcal{W}} t^2$.

Proof. Since $\Gamma; pc \vdash v_i : \tau_1$ and $\vdash \tau_1$ prot $\mathcal{H}$, BRACKET says $\Gamma; pc \vdash (\!| v_i |\!)_{\mathcal{H}} : \tau_1$. Lemmas 3.4 and 3.5 tell us that the result holds as stated above if and only if it holds when substituting $(\!| v_i |\!)_{\mathcal{H}}$ instead of v_i. From there we can apply Lemma 3.6. If there are no $(\downarrow^{\pi}_{\ell'},\ \overline{\eta}_{\ell}\ w)$ events in either trace with $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, then each $\mathcal{W}$-visible term in each trace must be equivalent to each other by induction on the number of elements (empty traces are equivalent). $\square$

Theorem 3.2 (Noninterference of high-pc programs)**.** *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H}$ be one of those high sets and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given an expression e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\ v_i \rangle \longrightarrow\!\!\!\!\!\rightarrow^{*} \langle v_i',\ t^i \rangle$ and $pc \in \mathcal{U} \cup \mathcal{S}$, then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

Proof. We claim that neither trace contains any $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_{\ell}\ w)$ elements where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, thus reducing this to Theorem 3.1. There are three cases to consider here: $pc \in \mathcal{H}$, $pc \in \mathcal{U}$ and $\mathcal{H} = \mathcal{S}$, and $pc \in \mathcal{S}$ and $\mathcal{H} = \mathcal{U}$.

Case $pc \in \mathcal{H}$**:** Both DECL and ENDORSE require $pc \sqsubseteq \ell$, so by upward-closure of $\mathcal{H}$, $\ell \in \mathcal{H}$.

Case $pc \in \mathcal{U}$ **and** $\mathcal{H} = \mathcal{S}$**:** DECL contains the condition $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})$. Converting into the authority lattice, this gives us $\ell^{\rightarrow} \wedge \Delta((\ell' \vee pc)^{\leftarrow}) \Rightarrow \ell'^{\rightarrow}$. Since $\mathcal{U} = \mathcal{A}^{\leftarrow}$ and $\mathcal{A}^{\leftarrow}$ is upward-closed, $pc \in \mathcal{U}$ means $(\ell' \vee pc)^{\leftarrow} \in \mathcal{A}^{\leftarrow}$. Condition 4 of Definition 3.6 then ensures $\Delta((\ell' \vee pc)^{\leftarrow}) \in \mathcal{A}^{\rightarrow}$. Because $\mathcal{A}^{\rightarrow}$ is an upward-closed sublattice, if $\ell^{\rightarrow} \in \mathcal{A}^{\rightarrow}$, then $\ell'^{\rightarrow} \in \mathcal{A}^{\rightarrow}$. Since $\mathcal{H} = \mathcal{S} = \mathcal{L} \setminus (\mathcal{A}^{\rightarrow})$, this means $\ell \notin \mathcal{H}$ only if $\ell' \notin \mathcal{H}$.

Case $pc \in \mathcal{S}$ **and** $\mathcal{H} = \mathcal{U}$**:** This argument is nearly identical to the previous case. ENDORSE means $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow})$ which, in the trust lattice, means $\ell'^{\leftarrow} \Rightarrow \ell^{\leftarrow} \vee \nabla((\ell' \wedge pc)^{\rightarrow})$. In the trust ordering, $\mathcal{A}$ is upward-closed, and thus $\mathcal{S} = \mathcal{L} \setminus (\mathcal{A}^{\rightarrow})$ is downward-closed. Since $pc \in \mathcal{S}$, that means $(\ell' \wedge pc) \in \mathcal{S}$. Condition 4 of Definition 3.6 then ensures $\nabla((\ell' \wedge pc)^{\rightarrow}) \notin \mathcal{A}^{\leftarrow} = \mathcal{H}$. Because $\mathcal{L} \setminus \mathcal{H}$ is a doward-closed sublattice, if $\ell^{\leftarrow} \notin \mathcal{H}$, then $\ell'^{\leftarrow} \notin \mathcal{H}$, as desired. □

Theorem 3.3 (Noninterference of secret–untrusted data)**.** *Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$. Let $\mathcal{H} = \mathcal{U} \cap \mathcal{S}$ and $\mathcal{W} = \mathcal{L} \setminus \mathcal{H}$. Given an expression e such that $\Gamma, x{:}\tau_1; pc \vdash e : \tau_2$ where $\vdash \tau_1$ prot $\mathcal{H}$, for all v_1, v_2 with $\Gamma; pc \vdash v_i : \tau_1$, if $\langle e[x \mapsto v_i],\ v_i\rangle \longrightarrow\!\!\!\!\rightarrow^{*} \langle v_i',\ t^i\rangle$ then $t^1 \approx^{\star}_{\mathcal{W}} t^2$.*

Proof. First we note that $\mathcal{H}$ is a high set as the intersection of two upward closed sets is also upward closed. We claim that neither trace contains any $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_{\ell}\ w)$ elements where $\ell' \in \mathcal{H}$ and $\ell \notin \mathcal{H}$, thus reducing this to Theorem 3.1. We will cover the two cases of π separately despite their similarities.

Case $\pi = \rightarrow$: The only way to emit $(\downarrow^{\rightarrow}_{\ell'}, \overline{\eta}_{\ell}\ w)$ is through E-DECL. By Theorem 3.8, each intermediate expression type-checks under the same initial context. This is a sub-expression, meaning there is some Γ', pc', and τ such that

$$\Gamma'; pc' \vdash \mathsf{decl}\ (\overline{\eta}_{\ell'}\ v)\ \mathsf{to}\ \ell : \ell\ \mathsf{says}\ \tau.$$

Therefore DECL ensures that $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc')^{\leftarrow})$. Every typing rule either moves pc up the information-flow lattice (by joining it with another label) or leaves it unchanged, so $pc \sqsubseteq pc'$ and thus $\Delta(pc'^{\leftarrow}) \sqsubseteq \Delta(pc^{\leftarrow})$, meaning $\ell'^{\rightarrow} \sqsubseteq \ell^{\rightarrow} \sqcup \Delta((\ell' \sqcup pc)^{\leftarrow})$. Converting to the authority lattice gives us $\ell^{\rightarrow} \wedge \Delta((\ell' \vee pc)^{\leftarrow}) \Rightarrow \ell'^{\rightarrow}$.

If $\ell' \in \mathcal{H}$, then $\ell'^{\rightarrow} \in \mathcal{S}$, and $\mathcal{S}$ is downward-closed in the trust ordering. Therefore $\ell^{\rightarrow} \wedge \Delta((\ell' \vee pc)^{\leftarrow}) \in \mathcal{S}$.

However, $\ell' \in \mathcal{H}$ also means $\ell'^{\leftarrow} \in \mathcal{U} = \mathcal{A}^{\leftarrow}$. Because $\mathcal{U}$ is upward-closed, $(\ell' \vee pc)^{\leftarrow} \in \mathcal{U}$. So by Condition 4 of Definition 3.6, $\Delta((\ell' \vee pc)^{\leftarrow}) \in \mathcal{A}^{\rightarrow} = \mathcal{L} \setminus \mathcal{S}$. Because $\mathcal{A}^{\rightarrow}$ is a sublattice, if $\ell^{\rightarrow} \in \mathcal{A}^{\rightarrow}$, then it would be the case that $\ell^{\rightarrow} \wedge \Delta((\ell' \vee pc)^{\leftarrow}) \in \mathcal{A}^{\rightarrow} = \mathcal{L} \setminus \mathcal{S}$, contradicting the above conclusion. Therefore $\ell^{\rightarrow} \in \mathcal{S}$. Since $\ell'^{\leftarrow} \in \mathcal{U}$ and DECL ensures $\ell^{\leftarrow} = \ell'^{\leftarrow}$, this means $\ell \in \mathcal{S} \cap \mathcal{U} = \mathcal{H}$, as desired.

Case $\pi = \leftarrow$: Dual to the above case, $(\downarrow^{\leftarrow}_{\ell'}, \overline{\eta}_{\ell}\ w)$ must be emitted from E-ENDORSE, so Theorem 3.8 now implies that there is some Γ', pc', and τ such that

$$\Gamma'; pc' \vdash \mathsf{endorse}\ (\overline{\eta}_{\ell'}\ v)\ \mathsf{to}\ \ell : \tau.$$

Since $pc \sqsubseteq pc'$, ENDORSE and the same logic as above require $\ell'^{\leftarrow} \sqsubseteq \ell^{\leftarrow} \sqcup \nabla((\ell' \sqcup pc)^{\rightarrow})$, which converts to $\ell'^{\leftarrow} \Rightarrow \ell^{\leftarrow} \vee \nabla((\ell' \wedge pc)^{\rightarrow})$ in the trust lattcie.

If $\ell' \in \mathcal{H}$, then $\ell'^{\leftarrow} \in \mathcal{U}$, and $\mathcal{U}$ is upward-closed in the trust ordering. Therefore $\ell^{\leftarrow} \vee \Delta((\ell' \wedge pc)^{\rightarrow}) \in \mathcal{U}$.

However, $\ell' \in \mathcal{H}$ also means $\ell'^{\rightarrow} \in \mathcal{S} = \mathcal{L} \setminus \mathcal{A}^{\rightarrow}$. Because $\mathcal{S}$ is downward-closed in the trust ordering, $(\ell' \wedge pc)^{\rightarrow} \in \mathcal{S}$. So by Condition 4 of Definition 3.6, $\nabla((\ell' \wedge pc)^{\rightarrow}) \notin \mathcal{A}^{\leftarrow} = \mathcal{U}$. Because $\mathcal{L} \setminus \mathcal{U}$ is a sublattice, if $\ell^{\leftarrow} \notin \mathcal{U}$, then it would be the case that $\ell^{\leftarrow} \vee \nabla((\ell' \wedge pc)^{\rightarrow}) \notin \mathcal{U}$, contradicting the above conclusion. Therefore, $\ell^{\leftarrow} \in \mathcal{U}$. Since $\ell'^{\rightarrow} \in \mathcal{S}$ and ENDORSE ensures $\ell^{\rightarrow} = \ell'^{\rightarrow}$, this means $\ell \in \mathcal{S} \cap \mathcal{U} = \mathcal{H}$, as desired.

Thus we see that for any trace element $(\downarrow^{\pi}_{\ell'}, \overline{\eta}_{\ell}\ w)$, if $\ell' \in \mathcal{H}$, then $\ell \in \mathcal{H}$, so by Theorem 3.1 the result holds. □

Theorem 3.6 (Nonmalleable information flow). *For any program e such that $\Gamma, x{:}\tau_x, y{:}\tau_y; pc \vdash e : \tau$, e enforces NMIF.*

Proof. We provide here only a proof of case 1 of Definition 3.5. All statements in case 2 are exactly dual so a precisely dual argument holds.

Let $\mathcal{A}$ be an attacker inducing high sets $\mathcal{U}$ and $\mathcal{S}$ and let $\mathcal{T} = \mathcal{L} \setminus \mathcal{U}$, $\mathcal{P} = \mathcal{L} \setminus \mathcal{S}$, and $\mathcal{W} = \mathcal{T} \cap \mathcal{P}$. To prove case 1 we prove a contrapositive of the stated implication. Specifically, if $t^{12}_{..n_{12}} \not\approx_{\ell} \mathcal{P}^{\star} t^{22}_{..n_{22}}$ but $\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1)$, then $t^{11}_{..n_{11}} \not\approx_{\ell} \mathcal{P}^{\star} t^{21}_{..n_{21}}$.

First we note that the theorem is uninteresting unless $\vdash \tau_x$ prot $\mathcal{S}$ and $\vdash \tau_y$ prot $\mathcal{U}$. Since v_i and w_j are both present in trace t^{ij}, if $v_1 \not\approx_{\mathcal{P}} v_2$ or $w_1 \not\approx_{\mathcal{T}} w_2$, the theorem is trivially true. In the former case, varying the first input clearly results in non-equivalent traces for both attacks. In the latter case, the result holds for $n_{ij} = 1$ and the precondition is clearly false otherwise. Now consider the case where $v_1 \approx_{\mathcal{P}} v_2$ and $w_1 \approx_{\mathcal{T}} w_2$. If $\vdash \tau_x$ prot $\mathcal{S}$ and $\vdash \tau_y$ prot $\mathcal{U}$, then these are trivially true—the interesting case we will handle below. Otherwise the equivalences require the contents of the value to be identical (except for nested secret/untrusted values). Any identical values clearly cannot result in distinguishable traces. We can handle nested secret/untrusted values by fixing the public/trusted parts of the inputs and viewing those values as the inputs instead, thus reducing to the case where

$\vdash \tau_x$ prot $\mathcal{S}$ and $\vdash \tau_y$ prot $\mathcal{U}$.[5] The rest of the proof assumes we are in this case.

By assumption $t^{12}_{..n_{12}} \not\approx_\ell \mathcal{P}^\star t^{22}_{..n_{22}}$, there is some point at which the traces become distinguishable. This can happen for one of two reasons: one trace is a prefix of the other (up to extra $\bullet$-equivalent terms), or there are two non-$\bullet$ terms that are non-equivalent.

In the first case, without loss of generality assume $t^{12}_{..n_{12}}$ is a prefix of $t^{22}_{..n_{22}}$. We claim that $t^{11}_{..n_{11}}$ is a prefix of $t^{21}_{..n_{21}}$. Lemma 3.6 ensures that any difference must come from a downgrade, which must be a declassification because ENDORSE prohibits changing the confidentiality and $\mathcal{P}$ allows labels of any integrity. By the argument in Theorem 3.3 this declassification event must be on trusted data, thus producing a public–trusted output. Thus the condition that $t^{i1}_{..n_{i1}-1} \approx^\star_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ ensures that any declassifications appearing in $t^{21}_{..n_{21}-1}$ appear identically in $t^{11}_{..n_{11}-1}$, and similarly for $t^{22}_{..n_{22}-1}$ and $t^{12}_{..n_{12}-1}$. Since $t^{ij}_{n_{ij}} \not\approx_{\mathcal{W}} \bullet$, any public–untrusted events appearing in $t^{2j}_{..n_{2j}}$ must also appear in $t^{1j}_{..n_{1j}}$, and thus $t^{11}_{..n_{11}}$ must be a prefix of $t^{21}_{..n_{21}}$.

For the second case—two non-$\bullet$ terms are non-equivalent—let n'_{i2} be the indices of the first such terms in each trace. That is, $t^{i2}_{n'_{i2}} \not\approx_{\mathcal{P}} \bullet$ and $t^{12}_{n'_{12}} \not\approx_{\mathcal{P}} t^{22}_{n'_{22}}$, but $t^{12}_{..n'_{12}-1} \approx^\star_{\mathcal{P}} t^{22}_{..n'_{22}-1}$. Again Lemma 3.6 ensures that the first non-equivalence must be the result of a declassification that exists in both traces.

By construction $n'_{ij} \le n_{ij}$ and we have that $t^{12}_{..n'_{12}-1} \approx^\star_{\mathcal{P}} t^{22}_{..n'_{22}-1}$ by the definition of n'_{ij} and $t^{i1}_{..n_{i1}-1} \approx^\star_{\mathcal{T}} t^{i2}_{..n_{i2}-1}$ by assumption. Consequently, for all $i, j, k, l \in \{1, 2\}$, we have

$$t^{ij}_{..n'_{ij}-1} \approx^\star_{\mathcal{W}} t^{kl}_{..n'_{kl}-1}.$$

In particular, this is true for t^{11} and t^{21}. We also know that for all $i, j \in \{1, 2\}$ $t^{ij}_{n'_{ij}} = (\downarrow^{\rightarrow}_{\ell'}, \overline{\eta}_\ell\ w_{ij})$ for some value w_{ij}.

We know that $w_{12} \not\approx_{\mathcal{P}} w_{22}$, and we now claim that $w_{11} \not\approx_{\mathcal{P}} w_{21}$. To emit this value, E-DECL must have been applied, meaning there must have been a term $(\overline{\eta}_{\ell'}\ w_{ij})$ in

[5]This technically reduces to the many-input version of the theorem that we prove is equivalent in Section 3.11.

the preceding expression. Since such expressions are not in the source language—they can only be created by applications of E-UnitM, E-Decl, and E-Endorse—this expression must appear previously in each trace. We know that $\ell \in \mathcal{P} \cap \mathcal{T}$ and Decl requires that $\ell'^{\leftarrow} = \ell^{\leftarrow}$, so therefore $\ell' \in \mathcal{T}$. Since the prefixes prior to this event are trusted-equivalent when varying only attacks, $w_{i1} \approx_{\mathcal{T}} w_{i2}$ for $i = 1, 2$. By assumption $w_{12} \not\approx_{\mathcal{P}} w_{22}$ and this is the first difference in those traces. This means that if $w_{11} \approx_{\mathcal{P}} w_{21}$ then w_{12} and w_{22} must have differed only in untrusted public values nested inside the original declassification (i.e., $w_{12} \approx \bullet^{\rightarrow}_{\mathcal{A}} w_{22}$). Therefore if $w_{11} \approx_{\mathcal{P}} w_{21}$, then v_1, v_2, and w_2 are exactly the inputs needed in Definition 3.2 to demonstrate that w_1 is an irrelevant attack. We assumed that $\mathrm{rel}^{\leftarrow}_{\mathcal{A},e}(w_1)$, so $w_{11} \not\approx_{\mathcal{P}} w_{21}$ and therefore $t^{11}_{..n_{11}} \not\approx_{\ell} \mathcal{P}^{\star} t^{21}_{..n_{21}}$, proving our desired result. $\square$

CHAPTER 4

COMPOSITIONAL REENTRANCY SECURITY

Compositional security remains a fundamental concern for software security. Code might appear secure, yet expose vulnerabilities when it interacts with other code. Blockchain smart contracts offer multiple prominent recent examples of this problem [126, 127, 131], but other instances exist. JavaScript code is difficult to secure when running on the same web page as code from a different source [45, 76, 109]. Web browsers themselves have fallen victim to attacks when executing code on web pages [49, 50]. In these settings, securing code in isolation is not sufficient. Reasoning about the behavior of a combination of interacting systems, however, is notoriously difficult. This work therefore aims for a way to build software with *compositional* security guarantees, meaning the security of an entire system follows from the security of its components.

Complex control flow, and in particular reentrant executions, pose a fundamental challenge for compositional security. Developers are increasingly building applications from separate communicating services that may belong to different trust domains [64, 163]. In such architectures, one service waiting for another to respond must be prepared to handle separate incoming requests. These *reentrant calls* effectively interrupt the execution of the application and, if the developer is not careful, can catch it in an inconsistent state, creating security vulnerabilities [51].

Reentrancy security has received much more attention since July 2016, when the Decentralized Autonomous Organization (DAO)—an Ethereum smart contract intended to function as a distributed venture capital fund—lost $50 million in tokens to such an attack, making global news [131]. Since then, a variety of methods have emerged to analyze or eliminate reentrancy attacks [5, 47, 57, 75, 100], but vulnerabilities continue to appear. For example, a January 2019 audit uncovered a reentrancy vulnerability in the Uniswap decentralized exchange [48]. The attack leveraged a subtle interaction between two contracts that were secure in isolation, and a third malicious contract. The first contract implicitly assumed the second would not call the malicious contract. Because the interface could not specify this expectation, developers used the exchange for a token standard that allowed for such calls. This choice led to the theft of $25 million worth of tokens in April 2020 [127], over a year after the original vulnerability disclosure.

We follow our previous suggestion [38] and use a general language-based technique to obtain compositional security even in the presence of reentrant executions. We define and enforce security using a semantic specification of trust in the form of information flow labels. Information flow control (IFC) has long been an appealing technique for obtaining compositional security and has proven useful in practice [66]. IFC type systems can guide software development with compile-time checking and provably enforce strong security guarantees such as noninterference. But while IFC is a good starting point for compositional security, existing approaches break down in the presence of reentrancy. Standard IFC rules either reject useful, secure applications by blocking requests from untrusted sources, or they allow insecure applications that are vulnerable to reentrancy attacks. We extend standard IFC rules to define a secure type system that efficiently and provably prevents attacks, yet is expressive enough to build interesting applications.

This approach addresses fundamental shortcomings of existing solutions. Current stand-alone reentrancy analyses [5, 75, 100] are non-compositional. That is, analyzing two pieces of code separately might not yield useful guarantees about their combination—the exact failing that led to the Uniswap attack. These tools also focus specifically on blockchain smart contracts. While smart contracts have provided notable recent examples of reentrancy vulnerabilities, similar exploits appear elsewhere [49–51] and there is no reason to limit solutions. The focus on smart contracts and the absence of trust specifications forces the tools to rely on contract boundaries—a syntactic construct—as a proxy for semantic security boundaries. This choice leads to a reentrancy definition we call *object reentrancy* that can judge the security of two semantically equivalent implementations differently, merely because the code has different structure.

There exist other language-based approaches that provide compositional guarantees and consider reentrancy, but they are again smart-contract focused and use object-based reentrancy definitions. Moreover, some limit expressiveness by outlawing reentrancy entirely [47, 57], while others provide only heuristic reentrancy protection [20, 149, 150]. In addition, they universally assume that all code is written in the same language. This strong assumption clearly does not apply to open systems where anyone can submit code, like Ethereum contracts or JavaScript on web pages. Even in closed systems with controlled environments and known code, new code might need to interact with legacy applications that do not respect the language rules.

We address these shortcomings by defining a new general-purpose security type system that tracks the integrity of data and computation. In addition to providing standard IFC data security guarantees, the type system combines with a run-time mechanism to provably eliminate dangerous reentrancy while allowing safe reentrancy. The guarantees, moreover, continue to hold even when trusted code interacts with untrusted code that does not obey the same restrictions.

The remainder of the chapter is structured as follows:

- Examples in Section 4.1 show the complexity of reentrancy.
- Section 4.2 provides background on information flow control and exposes its failure to handle reentrancy.

- Section 4.3 presents a new definition of security in the presence of reentrancy.
- Section 4.4 defines SeRIF, a core calculus that eliminates insecure reentrancy by combining a static IFC type system with a dynamic locking mechanism.
- Section 4.5 shows formally that SeRIF enforces our formal, compositional security condition.
- Section 4.6 describes a prototype type checker implementation and our experience using it on realistic programs.
- Section 4.7 discusses related work in more detail.
- Sections 4.8 – 4.12 include full details of SeRIF and all proofs.

4.1 Motivating Examples

By their very nature, reentrancy vulnerabilities are often hard to spot. For instance, the attack on Ethereum's Decentralized Autonomous Organization (DAO) was considered subtle at the time [52], despite being one of the simplest examples of reentrancy. To build intuition, we present three running examples of applications with reentrancy. Though we have distilled them to their core components, the vulnerabilities have undermined security in real-world applications.

4.1.1 Uniswap

We begin with the Uniswap/Lendf.me reentrancy vulnerability first identified in January 2019 [48] and later exploited in April 2020 [127]. The vulnerability arises from the combination of two contracts. Though each may be considered secure in isolation, they combine in unexpected ways, demonstrating the need for *compositional* reentrancy security.

```
contract Uniswap {
  Token tX, tY;

  function sellXForY(uint xSold) returns uint {
    uint prod = tX.getBal(this) * tY.getBal(this);
    uint yKept = prod / (tX.getBal(this) + xSold);
    uint yBought = tY.getBal(this) - yKept;

    assert tX.transferTo(msg.sender, this, xSold);
    assert tY.transferTo(this, msg.sender, yBought);
    return yBought;
  }
}

contract Token {
  function transferTo(address from, address to,
        uint amount) returns bool {
    ... // check and update balances

    from.alertSend(to, amount);
    to.alertReceive(from, amount);
    return true;
  }
}
```

Figure 4.1: Distilled Solidity [155] code for the Uniswap bug.

Uniswap is a smart contract platform where users can exchange one token for another. Figure 4.1 shows a simplified portion of the Uniswap contract: the exchange function `sellXForY` allows users to sell tokens of type X for tokens of type Y. Uniswap determines the exchange rate by the amount of X and Y it currently holds. It holds the product of the two amounts constant, allowing Uniswap to maintain the same total asset value as exchange rates fluctuate. The tokens themselves are implemented by independent contracts.

To perform an exchange, Uniswap first queries its balance with each token, then computes how much of token Y the user bought, and finally transfers tokens by calling `transferTo` on each token contract. Tokens execute transfers by first checking and updating balances, and then notifying the sender and recipient, allowing each in turn to execute arbitrary code.

Both contracts appear secure in isolation, following the best-practice recommendation of modifying state before making external calls to avoid reentrancy concerns [156]. However, when combined, they expose a dangerous exploit. Suppose the exchange begins with 6 units each of X and Y.

1. An attacker $\mathcal{A}$ calls `sellXForY` selling 6 units of X.
2. Uniswap correctly computes `prod` = 36 and `yBought` = 3.
3. Uniswap calls token X to transfer 6 units from $\mathcal{A}$.
4. The token notifies $\mathcal{A}$, giving it control of the execution.
5. Before returning, $\mathcal{A}$ calls `sellXForY` again to sell 6 more units of X, reentering the Uniswap contract.
6. Uniswap now has 12 units of X, but still 6 units of Y, so it computes `prod` = 72, not 36, and `yBought` = 2.

When the dust settles, Uniswap has 18 units of X and only 1 unit of Y, having given $\mathcal{A}$ an extra unit of Y and having broken the invariant that the product of the balances is 36. If desired, $\mathcal{A}$ can reclaim their original 12 units of X for only 2 units of Y, keeping the other 3 as illicit profit.

The fundamental problem is a mismatch between Uniswap's notion of secure behavior and the token's. The token correctly checks that all transfers are valid and authorized and follows programming patterns that avoid (internal) reentrancy concerns. No user can transfer more tokens than they have. Uniswap, however, implicitly assumes that `transferTo` transfers tokens and returns *without allowing an adversary to call Uniswap before it reestablishes the invariant that* `prod` *= 36.*

This insight suggests two approaches to fixing the bug: (1) token contracts

```
getOrCompute(key, computeFun) {
    i = _getIdx(key) // index of mapping if it exists
    if (mappings[i] == null) {
        mappings[i] = computeFun();
    }
    return mappings[i];
}
```

Figure 4.2: The `getOrCompute` function of a key–value store. Here `mappings` is an array that the store resizes as mappings are added.

could respect Uniswap's assumption by not calling unknown, untrusted code, or (2) Uniswap could stop relying on the assumption. Current platforms provide no way to guarantee the first option. Uniswap could state its assumption in documentation, but there is no technical means of specifying or enforcing it. Tokens that violate it could continue to freely interface with Uniswap, with disastrous results. The exchange can, however, implement the second option by acquiring a run-time lock on entry to the contract. It could then recognize the above attack and produce an error at step 5.

Our approach detects this vulnerability and can specify and correctly analyze either proposed solution. Among existing tools, only Nomos [57] can express the assumption in (1), which it mandates to statically eliminate all reentrancy. Other tools either cannot properly secure the application [20, 149, 150] or force the use of computationally expensive dynamic locks even when they are unnecessary [5, 47].

4.1.2 Key–Value Store

Smart contracts have made reentrancy concerns highly visible, but reentrancy is not unique to that domain. It has led to multiple critical security vulnerabilities in Internet Explorer [49, 50], and is a known concern for any application executing user-provided code [51].

For example, key–value stores often compute missing mappings with user-supplied functions [125, 143]. A careless implementation of this functionality can enable dangerous reentrancy. Consider the code in Figure 4.2, along with a `clear` method that frees `mappings` and installs a new empty array. An attacker can call `getOrCompute`, providing as arguments an unmapped key and a malicious function that calls `clear` and then returns a value. First `getOrCompute` computes `i`, then it calls the malicious function, which calls `clear` and replaces the `mappings` array. Finally `getOrCompute` attempts to write the attacker-provided value into index `i` of the new array.

If `i` is large—which is likely if the store previously contained many mappings—the write would be past the end of the new empty array. In languages like C/C++ without array bounds checking, an attacker-provided value would thus be written into an arbitrary memory location, enabling remote code execution or other critical security vulnerabilities. Even memory-safe languages like Java explicitly recommend developers check for reentrant modifications and throw exceptions [125].

Notably, while this attack appears very similar to concurrent-modification attacks on key–value stores, it requires no concurrency. Single-threaded applications or applications using simple thread-level locking are still vulnerable.

4.1.3 Town Crier

Banning all reentrancy might seem appealing, but this solution would be overly restrictive. Town Crier (TC) [182] is an example where safe reentrancy enables important functionality. TC provides authenticated data to smart contracts upon request. Users place requests with a smart-contract front end, and TC processes them asynchronously and delivers the data to user-specified callbacks when it is available. TC also allows users to cancel pending requests for a refund. Figure 4.3

```
 1 contract TownCrier {
 2   address[] requesters, callbacks;
 3
 4   function deliver(uint reqId, bytes data) {
 5     if (msg.sender == SERVICE_ADDR
 6         && requesters[reqId] != 0) {
 7       requesters[reqId] = 0;
 8       SERVICE_ADDR.call{value: FEE}("");
 9       callbacks[reqId].call(bytes);
10     }
11   }
12
13   function cancel(uint reqId) {
14     if (msg.sender == requesters[reqId]) {
15       requesters[reqId] = 0;
16       msg.sender.call{value: FEE}("");
17     }
18   }
19 }
```

Figure 4.3: Solidity [155] code for simplified partial Town Crier contract. Here `SERVICE_ADDR` is TC's trusted wallet address, and `FEE` is the request fee.

shows simplified versions of TC's `deliver` and `cancel` methods.

Invoking a user-provided callback in `deliver` opens the possibility of reentrant calls. Unlike in the previous examples, however, these calls are safe. By ensuring that the request status is always updated (lines 7 and 15) before calling untrusted code (lines 9 and 16), TC prevents attackers from receiving refunds for canceling requests that are mid-delivery or already canceled. Honest users, however, can still respond to data from one request by creating or canceling *different* requests.

For instance, a user contract may ask TC to function as a real-world timer and alert it at a specific real-world time. When woken up, the contract may determine that it needs to wait longer and request that TC send another alert, say, 2 hours later. A different user could make multiple parallel requests to retrieve the same data, e.g., a stock price, from several sources. Once enough responses have arrived, the user might wish to cancel the outstanding requests to reduce costs. Both of these patterns require safe reentrant calls into TC. This work aims to allow this *secure* reentrancy while still eliminating the vulnerabilities described above.

4.2 Information Flow Control

To obtain compositional security, it is natural to build on top of information flow control (IFC), a classic way to obtain compositional security guarantees such as noninterference [72]. Most IFC work has focused on data confidentiality [144, 175], but IFC can also protect integrity [19, 178] and availability [183]. As our goal is to guard against attackers performing unexpected calls into trustworthy code, we track only integrity.

IFC systems assign labels to computation and data within a system. As information flows through the system, the label on the destination of information is constrained to be no less restrictive than the label on its source. Since our goal is to enforce integrity, less trusted information should be prevented from influencing more trusted information.

Secure information flow is statically enforceable by a type system [144]. When linking separate code modules together, the security guarantees offered by the type system are automatically compositional, as long as the linked modules agree on types at interface boundaries and account for the confidentiality and integrity of the code itself [10]. Of course, real-world systems often have to interact with user-provided code or legacy applications that do not obey the rules of the type system. As we show, such noncompliant code can only violate the security guarantees of code that expresses trust in it.

4.2.1 Label model

We specify integrity using a set of integrity labels $\mathcal{L}$ and give each piece of data x a label ℓ_x representing its trust level. The labels have a reflexive, transitive relation $\ell_1 \Rightarrow \ell_2$, which we read "$\ell_1$ *acts for* ℓ_2," to denote that ℓ_1 is at least as trusted as ℓ_2. That is, anything that can influence data labeled ℓ_1 can also influence data labeled ℓ_2.[1] Data x can thus safely influence data y only when $\ell_x \Rightarrow \ell_y$. Influence can be either *explicit*—by assigning x directly to y—or *implicit*—by condition on x and assigning different values to y in each branch. For explicit flows, a simple check that $\ell_x \Rightarrow \ell_y$ at the point of assignment is sufficient. To control implicit flows, a *program counter label*, written pc, tracks the integrity of the computation itself, as is standard [144]. Inside a branch conditioned on x, the value of x has influenced control flow, so we require the constraint $\ell_x \Rightarrow pc$. Assigning a variable y to some value then requires $pc \Rightarrow \ell_y$, ensuring transitively that $\ell_x \Rightarrow \ell_y$.

$\mathcal{L}$ must also have some additional structure. Any pair of labels ℓ_1 and ℓ_2 must have a join, denoted $\ell_1 \vee \ell_2$, and a meet, denoted $\ell_1 \wedge \ell_2$. The join is the least upper bound and the meet is the greatest lower bound, so

$$\begin{aligned} \ell_1 \vee \ell_2 \Rightarrow \ell \quad &\Longleftrightarrow \quad \ell_1 \Rightarrow \ell \text{ and } \ell_2 \Rightarrow \ell \\ \ell \Rightarrow \ell_1 \wedge \ell_2 \quad &\Longleftrightarrow \quad \ell \Rightarrow \ell_1 \text{ and } \ell \Rightarrow \ell_2. \end{aligned}$$

We can then safely label information influenced by both ℓ_1 and ℓ_2 with label $\ell_1 \vee \ell_2$, for example. Lastly, join and meet must distribute: $\ell_1 \vee (\ell_2 \wedge \ell_3) = (\ell_1 \vee \ell_2) \wedge (\ell_1 \vee \ell_3)$. These properties collectively make $(\mathcal{L}, \Rightarrow)$ a *distributive lattice*.

This additional structure is only necessary to support our precise and flexible approach to enforcing reentrancy security. Luckily, existing label models are

[1] Most IFC systems use *flows-to*, denoted $\sqsubseteq$. We use acts-for as we find it intuitive, and the two mean the same thing when only tracking integrity.

typically distributive lattices, including two-point lattices, subset lattices of permissions [180], and free distributive lattices over a set of principals [11, 118]. In smart-contract systems, for example, it is natural to view contracts themselves as principals with different trust relationships among them. We might then employ *decentralized* information flow control [119] where labels are constructed from principals (e.g., contracts) that can influence data or computation.

4.2.2 Endorsement

Strictly enforcing IFC allows systems to enforce strong security properties like noninterference, which forbids *any* influence from untrusted information to trusted information. Noninterference, however, is too restrictive to build real applications, so practical IFC systems allow *downgrading*. Downgrading integrity, known as *endorsement* [185], treats information with a low-integrity label as being more trustworthy than its source would indicate.

From the IFC perspective, smart contracts and other similar services endorse frequently, though implicitly. They expose functions that accept calls from untrusted users, yet modify trusted local state. In other words, untrusted state affects trusted state, which an IFC system should only allow via endorsement.

Existing IFC languages support these trusted functions, but make them explicit. For example, the Jif language [102] supports *autoendorse* methods that can be called by an untrusted caller and that boost the integrity of the *pc* label on entry.

Viewed from the perspective of *pc* integrity, reentrancy attacks all exhibit a distinctive pattern: trusted (high-integrity) code calls lower-integrity code, which calls back into high-integrity code by exploiting endorsement. Existing endorsement mechanisms in Jif and other systems [65, 92, 98, 180] do not prevent this potentially dangerous control-flow pattern, and are thus vulnerable to reentrancy attacks. Preventing reentrancy attacks requires new restrictions on endorsement.

4.3 Reentrancy and Security

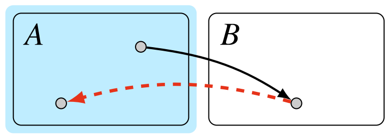
The examples in Section 4.1 show the need across application domains to constrain reentrancy without eliminating it entirely. We build on our previous work [38] to provide flexible definitions of reentrancy and security based on information flow control. This choice gives access to existing IFC tools and techniques with their strong data security guarantees, while making possible a precise, semantic specification of security.

4.3.1 Defining Reentrancy

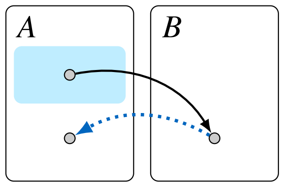
Prior work [5, 47, 75, 100] focuses on smart contracts and defines reentrancy in those terms: if contract A calls contract B, which calls back into contract A, the second call, and thus the entire execution, is considered reentrant. If no calls to A occur before the call to B returns, the execution is non-reentrant. We refer to this notion of reentrancy as *object reentrancy*, viewing contracts as a form of object.

We avoid object reentrancy because it relies on object boundaries—a fundamentally syntactic construct—to define security. Instead we define reentrancy with respect to the integrity level of computation. As integrity levels are part of a semantic security specification, using them to define a security-relevant property is sensible. This view leads to the following informal definition.

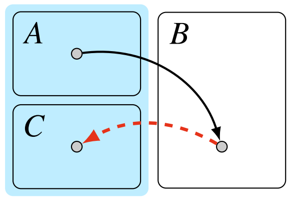
Definition 4.1 (ℓ-Reentrancy (informal))**.** If computation C_1 calls computation C_2, which then (possibly indirectly) calls C_3, the execution is reentrant with respect to label ℓ, or *ℓ-reentrant*, if C_1 and C_3 are trusted at ℓ, but C_2 is not.



(a) Object reentrancy and ℓ-reentrancy are the same when object and trust boundaries match.



(b) Partially-trusted objects can create object reentrancy that is not ℓ-reentrancy.



(c) Mutually-trusting objects can create ℓ-reentrancy that is not object reentrancy.

Figure 4.4: Comparing ℓ-reentrancy to object reentrancy. Boxes represent objects, the blue shaded region is high-integrity code, and arrows represent calls.

Note that C_1 and C_3 may be the same or different, as long as they are both trusted at ℓ.

Figure 4.4 depicts how ℓ-reentrancy relates to object reentrancy. If an entire object is trusted at ℓ and nothing else is (Figure 4.4a), ℓ-reentrancy and object reentrancy align. However, object and trust boundaries may differ, leading to different definitions. If a trusted operation in A calls untrusted B, a call to an *untrusted* portion of A (Figure 4.4b), would be considered reentrant in an object-based definition but not ℓ-reentrancy. Such a call could correspond to a Town Crier user updating a request callback during data delivery or a web app accessing untrusted user profile data while modifying a trusted billing key–value store. These operations are never dangerous, as low-integrity operations cannot damage high-integrity data. By contrast, one application may be split across multiple mutually trusting objects. Such a split in Ethereum's Parity Wallet led to two famous attacks [25, 126]. For an application split across A and C, if A calls B, then a call from B into C (Figure 4.4c) is a reentrant call into the application. By relying on trust levels, ℓ-reentrancy properly identifies this pattern as reentrancy, while object reentrancy does not.

To employ ℓ-reentrancy, each operation needs an integrity level. Conveniently, the *pc* label used to control implicit information flows (Section 4.2.1) provides such a label. It combines the integrity of the code and the integrity of data influencing the control flow to specify how trusted an operation is to execute when it does, making it ideal to define a property of trusted and untrusted operations calling each other.

4.3.2 Reentrancy Security

While ℓ-reentrancy defines reentrancy based on integrity patterns of the control flow, it does not tell us when it is secure. Some work [47, 57] simply declares all reentrancy (according to their definition) dangerous and to outlaws it entirely. With an appropriate definition of reentrancy, this approach eliminates vulnerabilities, but safe reentrancy has legitimate uses, as illustrated by the Town Crier example.

To eliminate the need for difficult manual reentrancy analysis, we define "secure reentrancy" as reentrancy that programmers can ignore when analyzing correctness. One way to accomplish this goal is to ensure that reentrancy cannot enable program behaviors that would not exist without it. These behaviors could be program invariants, such as Uniswap holding the product of its asset quantities constant or the key–value store never writing to unallocated memory; they could be statements about how state changes, like Town Crier's request ID monotonically increasing; or they could be more complex properties like noninterference.

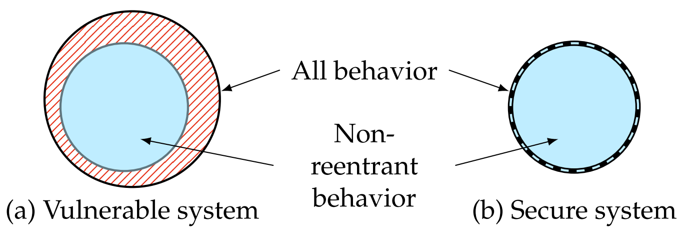


Figure 4.5: The set of possible behaviors in a secure vs. a vulnerable system. In a vulnerable system, reentrancy can introduce behaviors not possible without it. In a secure system, all behaviors are possible in non-reentrant executions.

Programmers cannot hope to guarantee properties that unknown or untrusted code can directly violate, so our definition ignores such properties entirely. Specifically, ℓ-reentrancy security considers only properties defined over state trusted at label ℓ. We refer to these as *ℓ-integrity properties*, leading to the following security definition, depicted visually in Figure 4.5.

Definition 4.2 (Reentrancy Security (informal))**.** A program is *ℓ-reentrancy-secure* if every ℓ-integrity property, such as a program invariant, that holds for all non-ℓ-reentrant executions holds for all executions.

Definition 4.2 specifies a semantic notion of security. It also provides a principled explanation for a common best practice for smart-contract programming: to perform all state modifications before calling other contracts [156]. Done properly, this design pattern ensures that any reentrant calls occur after all logic in the original execution has completed. As a result, those reentrant calls would have the same effect as making a second, non-reentrant call after the first call completes. Because this reentrancy stems from untrusted calls that the original trusted code made in tail position, we refer to it as *tail reentrancy*.

The definition is also flexible. For a specific application, we could refine it to require only that reentrancy does not violate particular programmer-specified application properties. To keep annotation burden low and to avoid the need to specify detailed program properties, our definition requires that ℓ-reentrant executions maintain *all* properties that hold without reentrancy. However, the later formal definition (Definition 4.9) allows such refinement simply by restricting a universal quantifier.

4.3.3 Enforcing Reentrancy Security

As described above, ℓ-reentrancy occurs when high-integrity code calls into low-integrity code that then calls back into high-integrity code before returning. IFC only permits this pattern through the autoendorse mechanism described in Section 4.2.2. Many services, including the examples in Section 4.1, require untrusted users to make requests into trusted code, making some version of autoendorse necessary. We therefore allow it, but with additional restrictions.

In particular, endorsement of control flow is restricted by *locking integrity*. When a function endorses the integrity of the control flow to label ℓ, integrity ℓ is locked, preventing further endorsement up to ℓ until the original call returns. Honest users can then invoke a service one or more times in sequence using a call-and-return pattern, but adversaries are unable to reenter into high-integrity code.

The semantics of these locks is to prevent autoendorsement from granting integrity that is locked. A trusted operation is then always given the chance to reestablish any high-integrity invariants or properties it may have temporarily invalidated before an attacker can invoke another trusted operation. To safely autoendorse from integrity pc_1 to integrity pc_2, for any operation pc_2 is trusted to perform, either pc_1 must already be trusted at that level or the requisite integrity must be unlocked. Formally, when integrity ℓ_L is locked, then for all labels ℓ, if $\ell_L \Rightarrow \ell$ and $pc_2 \Rightarrow \ell$, then $pc_1 \Rightarrow \ell$. The definition of lattice join quickly shows that this rule is equivalent to $pc_1 \Rightarrow pc_2 \vee \ell_L$.

We could track and enforce locks statically, as part of the type system, or dynamically in the runtime. Static locking—proving that a dynamic lock would never prevent execution—imposes no overhead and avoids unexpected errors at run time. Unfortunately, purely static locks interact poorly with code that may not enforce the same guarantees. Since any unknown code could call autoendorse functions—violating a static lock, meaning a dynamic lock would halt execution—a sound type system must assume the worst and prevent all calls to that code when integrity might be locked. This highly restrictive outcome would violate a core design goal of this work: providing compositional security even when interacting with unknown code. Dynamic locks avoid this constraining over-approximation at the expense of run-time cost.

We therefore take a hybrid approach and separate locked integrity into a static component and a dynamic one. The type system automatically adds endorsed control flow to the static component, but programmers can explicitly move integrity from the static component to the dynamic one. This approach achieves the run-time efficiency and predictability of static mechanisms when security can be proved statically, while still supporting safe interaction with unknown or untrusted code through more expressive dynamic locks.

The calculus does not specify how to implement dynamic locks. They could be built into the runtime, tracked by a security monitor, or even implemented as a library. So long as all code trusted at level ℓ is well-typed and agrees on *some* protocol to enforce the dynamic portion of the locks, the system will preserve ℓ-reentrancy security. There is no requirement that untrusted check integrity locks statically or dynamically.

$$
\begin{array}{rcl}
f, m, x & \in & \mathcal{V} \quad \text{(variable, method, and field names)} \\
\ell, pc & \in & \mathcal{L} \quad \text{(integrity labels)} \\
t & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{ref}\ \tau \mid C \\
\tau & ::= & t^{\ell} \\
CL & ::= & \mathsf{class}\ C[\ell]\ \mathsf{extends}\ C\ \{\overline{f}:\overline{\tau}\ ;\ K\ ;\ \overline{M}\} \\
K & ::= & C(\overline{f}:\overline{\tau})\ \{\mathsf{super}(\overline{f})\ ;\ \mathsf{this}.\overline{f} = \overline{f}\} \\
M & ::= & \tau\ m\{pc \gg pc; \ell\}(\overline{x}:\overline{\tau})\ \{e\} \\
v & ::= & x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \iota \mid \mathsf{null} \mid \mathsf{new}\ C(\overline{v}) \\
e & ::= & v \mid \mathsf{if}\{pc\}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{ref}\ v\ \tau \mid !v \mid v := v \\
 & \mid & (C)v \mid v.f \mid v.m(\overline{v}) \\
 & \mid & \mathsf{endorse}\ v\ \mathsf{from}\ \ell\ \mathsf{to}\ \ell \mid \mathsf{lock}\ \ell\ \mathsf{in}\ e \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e
\end{array}
$$

Figure 4.6: Syntax for SeRIF.

4.4 A Core Calculus for Secure Reentrancy

We present the Secure-Reentrancy Information Flow Calculus (SeRIF), an object-oriented core calculus that models how a programming language can implement the above ideas. Figure 4.6 gives the syntax for SeRIF. It extends Featherweight Java (FJ) [83] with information flow labels and, to support mutation, also reference cells [129, Chapter 13].

SeRIF employs fine-grained IFC, so each type τ consists of a base type t and an integrity label ℓ. For simplicity, we limit base types to unit, bool, references, and object types. To simplify proofs, null references are allowed.

Class and method definitions extend those in FJ with integrity labels. To model distributed systems, we consider code a form of data that may come from multiple sources, so each class definition CL includes a label ℓ_C for the integrity of the code.

A method definition M contains labels $pc_1 \gg pc_2; \ell$. Most IFC systems give functions a single pc label, but SeRIF has two: pc_1 specifies the minimum integrity required to call m, while pc_2 specifies the integrity at which m operates. Separating these labels supports autoendorsement as described in Section 4.2.2. If $pc_1 \not\Rightarrow pc_2$, then m is an autoendorse function. Both pc labels are bounded by ℓ_C, so code may only perform operations that ℓ_C is trusted to perform. The label ℓ specifies the locks method m promises not to violate.

The if syntax includes the pc label used for the branches. We make this label explicit only to simplify the operational semantics. In practice, it is easy to infer automatically.

The endorse expression endorses data as in other IFC systems with downgrading. The term lock ℓ in e converts static locks to dynamic ones. In the operational semantics, e executes with ℓ dynamically locked, so the type system can safely release any static lock on ℓ when type-checking e.

Expression subterms consist mostly of (open) values, not arbitrary expressions. In particular, let statements are the only way to sequentially compose computation.

Because SeRIF is object-oriented, it can model interacting services and reentrancy concerns. An application or contract implementation is a class, and a contract or instance of that application is an object of that class type, allowing easy interaction between different services. Moreover, inheritance allows applications that share common features to inherit form a common parent. For example, a blockchain smart contract system can be modeled by having all contracts inherit from a Contract class that implements tracking of currency.

4.4.1 SeRIF Operational Semantics

SeRIF has a small-step substitution-based semantics. Most rules are standard for an object-oriented language with mutable references [83, 129], with a few additions for security.

Because expressions are built mostly out of values, evaluation contexts are simple. Indeed, let expressions are the only surface syntax to serve as evaluation contexts. We introduce three new syntactic forms as evaluation contexts to enable precise tracking of function boundaries, execution integrity, and dynamic locks. These *statements* are denoted by s.

$$\begin{array}{rcl} E & ::= & [\cdot] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{return}_{\tau}\ E \mid E\ \mathsf{at\text{-}pc}\ pc \mid E\ \mathsf{with\text{-}lock}\ \ell \\ s & ::= & E[e] \end{array}$$

Semantic steps are defined on a pair of a statement s and a *semantic configuration*: a four-tuple $C = (CT, \sigma, \mathcal{M}, L)$. Unlike in FJ, the class table CT is explicit, as the security definitions in Section 4.5 quantify over possible class tables. A heap σ maps locations to value–type pairs, and Σ_{σ} denotes the location-to-type mapping induced by σ. That is, $\Sigma_{\sigma}(\iota) = \tau$ if and only if $\sigma(\iota) = (v, \tau)$ for some v. The final two elements, $\mathcal{M}$ and L are both lists of integrity labels. $\mathcal{M}$ tracks the integrity of executing code, and L tracks the dynamic portion of the currently-locked integrity. For notational ease, we reference the components of C freely when only one group is in scope and we write $C[X/L]$ to denote $(CT, \sigma, \mathcal{M}, X)$, and similarly for σ and $\mathcal{M}$.

Figure 4.7 presents selected semantic rules. The complete semantics is in Figure 4.9 (Section 4.8). In the semantic rules, v refers to a closed value, not a variable. In addition to many standard rules, the rules E-LOCK and E-UNLOCK dynamically lock and unlock labels. The semantics abstracts out the many possible lock implementations, merely tracking the set of locked labels and defining where to check them. The rules for conditionals (E-IFT and E-IFF) now include tracking terms.

The key rule is E-CALL. It looks up the definition of a method with *mbody* (Section 4.8) and performs several dynamic checks: it verifies that the arguments all have the correct types, that the caller has sufficient integrity to invoke the function, and that calling the method does not violate any dynamically locked label $\ell \in L$.

[E-IfT]
$$\frac{}{\langle \textsf{if}\{pc\}\ \textsf{true then}\ e_1\ \textsf{else}\ e_2 \mid C\rangle \longrightarrow \langle e_1\ \textsf{at-pc}\ pc \mid C\rangle}$$

[E-AtPc]
$$\frac{}{\langle v\ \textsf{at-pc}\ pc \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

[E-Ref]
$$\frac{\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \textsf{ref}\ v\ \tau \mid C\rangle \longrightarrow \langle \iota \mid C[\sigma[\iota \mapsto (v,\tau)]/\sigma]\rangle}$$

[E-Assign]
$$\frac{\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \iota := v \mid C\rangle \longrightarrow \langle () \mid C[\sigma[\iota \mapsto (v,\tau)]/\sigma]\rangle}$$

[E-Call]
$$\frac{\begin{array}{c} mbody(C,m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L}(pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \textsf{this} \mapsto \textsf{new}\ C(\overline{v})] \end{array}}{\langle \textsf{new}\ C(\overline{v}).m(\overline{w}) \mid C\rangle \longrightarrow \langle \textsf{return}_\tau\ (e'\ \textsf{at-pc}\ pc_2) \mid C[\mathcal{M}, \ell_m/\mathcal{M}]\rangle}$$

[E-Return]
$$\frac{\Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \textsf{return}_\tau\ v \mid C\rangle \longrightarrow \langle v \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

[E-Lock]
$$\frac{}{\langle \textsf{lock}\ \ell\ \textsf{in}\ e \mid C\rangle \longrightarrow \langle e\ \textsf{with-lock}\ \ell \mid C[L, \ell/L]\rangle}$$

[E-Unlock]
$$\frac{L = L', \ell}{\langle v\ \textsf{with-lock}\ \ell \mid C\rangle \longrightarrow \langle v \mid C[L'/L]\rangle}$$

Figure 4.7: Selected small-step semantic rules for SeRIF.

Dynamic Security Checks. The E-Ref, E-Assign, E-Call, and E-Return rules contain dynamic checks for type safety and information security. These checks prevent untrusted code from placing ill-typed values in the heap or passing them to trusted code. They similarly prevent untrusted code from modifying trusted heap locations in any way. Such checks are critical for trusted code to safely interact with ill-typed attacker code in any information flow system. While we do not detail how to implement dynamic typing or label checks here, there is considerable research into both. Gradually typed languages do run-time type checking [154], and distributed IFC systems include run-time label checks [e.g., 69, 98, 179]. Moreover, when all high-integrity code is well-typed, it is sufficient to isolate memory between objects, as in Ethereum contracts [173], and execute run-time checks when entering trusted code.

4.4.2 Type System for SeRIF

The type system for SeRIF contains two different forms for typing judgments: one for values and one for expressions. The typing judgment for values is straightforward for a stateful language. It takes the form $\Sigma; \Gamma \vdash v : \tau$ where Σ is a heap type mapping references to types and Γ is a typing environment mapping variables to types. We write $\Sigma \vdash v : \tau$ when Γ is empty, as we did in Section 4.4.1.

Values specify no computation so they need not reason about security. Typing judgments for expressions are more complex, including a standard *pc* label to track the integrity of the control flow. To secure reentrancy with static locks when possible, they also include a label ℓ representing locked integrity.

Allowing tail reentrancy while eliminating other forms of ℓ-reentrancy requires treating calls in tail position differently from calls in other positions. We accomplish this goal not by restricting when a given call can occur, but instead by restricting what can occur *after the call returns*. Instead of one lock label, this strategy uses two: an *input lock* ℓ_{I} that an expression must maintain to safely execute outside tail position, and an *output lock* ℓ_{O} specifying the locks the expression *actually* maintains. The typing judgment now takes the form $\Sigma; \Gamma; pc; \ell_{\mathrm{I}} \vdash e : \tau \dashv \ell_{\mathrm{O}}$.

For an expression e to type-check with input lock ℓ_{I}, each subexpression of e outside tail position must maintain ℓ_{I}. As non-value expressions only appear outside of tail position in let expressions, the following typing rule enforces this re-

$$[\textsc{If}]\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash v : \mathsf{bool}^{\ell} \qquad \ell \Rightarrow pc \qquad \ell \triangleleft \tau \\ \Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash e_1 : \tau \dashv \ell_{\mathrm{O}} \qquad \Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash e_2 : \tau \dashv \ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash \mathsf{if}\{pc\}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \dashv \ell_{\mathrm{O}}}$$

$$[\textsc{Assign}]\quad \frac{\begin{array}{c}\Sigma;\Gamma \vdash v_1 : (\mathsf{ref}\ \tau)^{\ell} \\ \Sigma;\Gamma \vdash v_2 : \tau \qquad \ell \triangleleft \tau\end{array}}{\Sigma;\Gamma;\ell;\ell_{\mathrm{I}} \vdash v_1 := v_2 : \mathsf{unit}^{\ell'} \dashv \ell_{\mathrm{O}}}$$

$$[\textsc{Call}]\quad \frac{\begin{array}{c} mtype(C,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2;\ell_{\mathrm{O}}} \tau_0 \qquad \Sigma;\Gamma \vdash v : C^{\ell} \qquad \Sigma;\Gamma \vdash \overline{v_a} : \overline{\tau_a} \\ \ell \Rightarrow pc_1 \qquad pc_1 \Rightarrow pc_2 \vee \ell_{\mathrm{I}} \qquad \tau_0 <: \tau \qquad pc_2 \vee \ell \triangleleft \tau\end{array}}{\Sigma;\Gamma;pc_1;\ell_{\mathrm{I}} \vdash v.m(\overline{v_a}) : \tau \dashv \ell_{\mathrm{O}} \vee pc_2}$$

$$[\textsc{Lock}]\quad \frac{\begin{array}{c}\Sigma;\Gamma;pc;\ell'_{\mathrm{I}} \vdash e : \tau \dashv \ell'_{\mathrm{O}} \\ \ell'_{\mathrm{I}} \wedge \ell \Rightarrow \ell_{\mathrm{I}} \qquad \ell'_{\mathrm{O}} \wedge \ell \Rightarrow \ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash \mathsf{lock}\ \ell\ \mathsf{in}\ e : \tau \dashv \ell_{\mathrm{O}}}$$

$$[\textsc{Method-Ok}]\quad \frac{\begin{array}{c}\ell_{\mathrm{I}} \Rightarrow pc_2 \qquad \ell_C \Rightarrow pc_2 \qquad \ell_{\mathrm{I}} \vee \ell'_{\mathrm{O}} \Rightarrow \ell_{\mathrm{O}} \qquad pc_1 \triangleleft \overline{\tau_a} \\ \Sigma; \overline{x}{:}\overline{\tau_a}, \mathsf{this}{:}C^{pc_2}; pc_2; \ell_{\mathrm{I}} \vdash e : \tau \dashv \ell'_{\mathrm{O}} \\ CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\} \\ (D,m) \in \mathrm{dom}(mtype) \implies mtype(D,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2;\ell_{\mathrm{O}}} \tau\end{array}}{\Sigma \vdash \tau\ m\{pc_1 \gg pc_2; \ell_{\mathrm{O}}\}(\overline{x}{:}\overline{\tau_a})\ \{e\}\ \mathsf{ok\ in}\ C}$$

Figure 4.8: Selected typing rules for SeRIF

striction.

$$[\textsc{Let}]\quad \frac{\begin{array}{c}\Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash e_1 : \tau_1 \dashv \ell'_{\mathrm{O}} \qquad \ell'_{\mathrm{O}} \Rightarrow \ell_{\mathrm{I}} \\ \Sigma;\Gamma, x{:}\tau_1;pc;\ell_{\mathrm{I}} \vdash e_2 : \tau_2 \dashv \ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \ell_{\mathrm{O}}}$$

This rule is standard except that it requires $\ell'_{\mathrm{O}} \Rightarrow \ell_{\mathrm{I}}$, capturing the intuition above: e_1 must maintain at least lock ℓ_{I}, as it is outside tail position. Because e_2 is in tail position in this expression, there is no similar restriction on ℓ_{O}.

Figure 4.8 contains selected typing rules for SeRIF. The notation $\ell \triangleleft \tau$ indicates that data of type τ is no more trusted than ℓ; that is, $\ell \triangleleft t^{\ell'}$ if and only if $\ell \Rightarrow \ell'$. The rules also use the auxiliary lookup functions *fields* and *mtype* as well as a subtyping relation $<:$ that includes standard object subtyping and safe relabeling—$t^{\ell} <: t^{\ell'}$ if and only if $\ell \Rightarrow \ell'$. The complete type system is in Figure 4.10 in Section 4.8.

Most typing rules (e.g., IF and ASSIGN) are standard for an information flow calculus [144]. The only non-standard rules are those that directly reference or constrain static locks: sequential composition (LET), method calls (CALL), and dynamic locking (LOCK).

Most of the premises of CALL are standard. They check that the object and arguments have appropriate types and ensure information-security of the return type and control flow of the call. They also check that the call does not violate any static locks ($pc_1 \Rightarrow pc_2 \vee \ell_\mathrm{I}$) and that it attenuates trust in the output by the integrity of both the object and the method ($pc_2 \vee \ell \triangleleft \tau$).

This rule has two notable features. The first is not what it requires, but rather what it *does not* require. There is no relation between the static input locks ℓ_I of the surrounding environment and ℓ_O, the locks maintained by the method itself. This lack of constraint is precisely what enables tail reentrancy. A call in tail position need not maintain any locks, so it may result in reentrancy. Outside tail position, however, the LET rule requires that the output locks of the call expression—bounded by the locks maintained by the method—must act for ℓ_I. CALL and LET therefore combine to enable safe tail reentrancy while ruling out other potentially dangerous reentrancy.

The second feature is that CALL does not maintain locks ℓ_O—the locks maintained by the method—but instead only $\ell_\mathrm{O} \vee pc_2$. This adjustment enables safe interaction with untrusted code that might not enforce the same guarantees as SeRIF. Such code may claim to maintain locks, but fail to do so. Our safeguard follows the principle of decentralized IFC [119]: you can only be hurt by an adversary you trust. We therefore attenuate the claimed lock label ℓ_O by the integrity of the code.

Due to SeRIF's inheritance structure, however, they type system cannot determine the exact integrity of the code. The implementation of m may come from C or any of its superclasses or subclasses. We instead need a bound on the implementation's integrity. The class typing rule METHOD-OK requires that the integrity of the code act for pc_2 to define or override a method with integrity pc_2. As a result, pc_2 is the most precise bound on the code's integrity available to the type system.

To understand the LOCK rule, recall that the lock term is designed to convert static locks to dynamic ones. The type system must ensure that ℓ_I, the previous input locks, remain locked in some manner, but it can safely release the portion that is dynamically checked. In particular, LOCK splits ℓ_I into ℓ and some ℓ'_I such that $\ell'_\mathrm{I} \land \ell \Rightarrow \ell_\mathrm{I}$. Now ℓ_I will remain locked as long as e type-checks with static input lock ℓ'_I. Similarly, lock ℓ in e actually maintains locks on both ℓ'_O—the locks e maintains—and ℓ. It is thus safe to trust ℓ_O up to $\ell'_\mathrm{O} \land \ell \Rightarrow \ell_\mathrm{O}$.

Finally, METHOD-OK defines when a method is well-typed. This rule implements the idea that autoendorse methods statically lock integrity by default. Specifically, it requires $\ell_\mathrm{I} \Rightarrow pc_2$, so any expression outside tail position must respect locks on the new, higher integrity of control flow. The integrity of the code must also act for the integrity with which the function executes ($\ell_C \Rightarrow pc_2$), ensuring code cannot do anything its source is not trusted to do. Next, the locks the method claims to enforce (ℓ_O) must be maintained both initially (ℓ_I) and throughout (ℓ'_O). The last information-security check ($pc_1 \triangleleft \overline{\tau_a}$) guarantees that any code trusted to call the method is also trusted to provide its arguments.

4.4.3 Modeling Application Operation

We aim to model applications that, like smart contracts, service user requests and may persist state across requests. We represent the current state of the world by a set of class definitions in a class table CT and a state map σ. A single user interaction, which we term an *invocation* I, is a label specifying the user's integrity and a call to a single method of an object stored in σ.

Execution of an invocation $I = (\iota, m(\overline{v}), \ell)$ with state σ starts from a semantic configuration with the expression, integrity ℓ, and no locks, and step it to completion. The notation $(I, CT, \sigma) \Downarrow \sigma'$ signifies that it terminates in updated state σ'. The following rule formalizes this idea, using $!\iota.m(\overline{v})$ as shorthand for $\mathsf{let}\ o = !\iota\ \mathsf{in}\ o.m(\overline{v})$.

$$[\text{E-INVOKE}]\ \frac{\langle !\iota.m(\overline{v}) \mid (CT, \sigma, \ell, \cdot)\rangle \longrightarrow^{*} \langle w \mid (CT, \sigma', \ell, \cdot)\rangle}{(I, CT, \sigma) \Downarrow \sigma'}$$

We use the same notation to denote running a list of invocations $\overline{I}$ in sequence, using the output state from one as the input state from the next. That is, when $\overline{I} = I_1, \ldots, I_n$ and $(I_i, CT, \sigma_{i-1}) \Downarrow \sigma_i$ for each $1 \leq i \leq n$, we write $(\overline{I}, CT, \sigma_0) \Downarrow \sigma_n$.

To type-check an invocation, the expression used in the evaluation must be well-typed in the evaluation environment:

$$[\text{INVOKE}]\ \frac{\Sigma; \cdot; \ell; \ell_{\mathrm{I}} \vdash !\iota.m(\overline{v}) : \tau \dashv \ell_{\mathrm{O}}}{\Sigma \vdash (\iota, m(\overline{v}), \ell)}$$

4.4.4 Examples Revisited

We now revisit the examples from Section 4.1 to see how SeRIF detects application vulnerabilities while permitting secure implementations.

Uniswap. The vulnerability (Section 4.1.1) stems from an unexpected interaction between an exchange, tokens, and a malicious user. While they may all have different integrity, for simplicity, we give the exchange and the tokens the same trusted label T and the user an untrusted label U with $U \not\Rightarrow T$.

Anyone can call `sellXForY`, but it computes how much of asset Y to move and transfers tokens, so it must have label $U \gg T; \ell_{\mathrm{O}}$ for some ℓ_{O}. Similarly, the token's `transferTo` method modifies high-integrity records, so it needs label $pc \gg T; \ell'_{\mathrm{O}}$ for some labels pc and ℓ'_{O}.

The METHOD-OK rule requires `sellXForY` to type-check with some ℓ_{I} where $\ell_{\mathrm{I}} \Rightarrow T$. Because we sequence two calls to `transferTo`, LET requires either a dynamic lock on label T around (at least) the first transfer. or $\ell'_{\mathrm{O}} \Rightarrow \ell_{\mathrm{I}} \Rightarrow T$. These options correspond precisely to the solutions suggested in Section 4.1.1. Requiring $\ell'_{\mathrm{O}} \Rightarrow T$ is a statement that Uniswap expects the tokens not to call untrusted code. A dynamic lock, by contrast, secures the exchange without assuming any particular token behavior and correspondingly allows any value of ℓ'_{O}.

Notably, `transferTo` can type-check with $\ell'_{\mathrm{O}} \Rightarrow T$ in either of two ways: it can decline to call unknown code (i.e., remove lines 20 and 21 in Figure 4.1), or the token itself could acquire a dynamic lock while making the calls. The first option straightforwardly eliminates the vulnerability. By dynamically locking T, the second option will prevent an attacker from making reentrant calls to either the token or the exchange during a transfer.

Key–value store. We use the same labeling scheme: the key–value store application gets a trusted label T while the user gets an untrusted label U. Because anyone can call `getOrCompute` but it modifies trusted data, it must have label $U \gg T; \ell_{\mathrm{O}}$ for some ℓ_{O}. The user-provided computation function is not trusted, so it gets label $pc \gg U; \ell'_{\mathrm{O}}$ for some labels pc and ℓ'_{O}.

As in the Uniswap example above, METHOD-OK requires `getOrCompute` to type-check with some $\ell_{\mathrm{I}} \Rightarrow T$. Because the user-provided fallback function executes in sequence before another trusted operation, LET and CALL combine to require either a dynamic lock or $\ell'_{\mathrm{O}} \vee U \Rightarrow \ell_{\mathrm{I}} \Rightarrow T$. This second option, however, is impossible because $U \not\Rightarrow T$.

This forced reliance on a dynamic lock stems from the type system not trusting the user-provided callback to even type-check. In a modified type system that separated trust in the code's execution from trust that it type-checks, it would be sufficient to require that it type-check with high-integrity and some $\ell'_O \Rightarrow T$. This solution would correspond to a static guarantee that the user-provided callback does not invoke `clear` or any other method modifying the store's internal state.

Town Crier. As described in Section 4.1.3 and the original paper [182], Town Crier is secure despite using (object) reentrancy, and the type system can verify that. Using the same labels again, we label Town Crier and the trusted service address T and the user U. We can give the functions the following signatures.

```
int request{U≫T;T}(params:t^U, callback:address^U)
void cancel{U≫T;U}(id:int^U)
void deliver{T≫T;U}(id:int^T, data:bytes^T)
```

The `request` method—which just records the request parameters and updates a counter—type-checks simply. The `cancel` method type-checks with an endorsement on the condition on line 14 of Figure 4.3. Type-checking `deliver` relies on TC trusting `SERVICE_ADDR` not to call attackers when receiving money. However, `SERVICE_ADDR` is a hard-coded wallet address with no code that is already trusted to provide data to `deliver`, so the operation sending it money can safely have the signature $T \gg T; T$. These labels allow `deliver` to type-check as written.

4.5 Formalizing Security Guarantees

We now have the tools needed to formalize reentrancy and security definitions from Section 4.3.

4.5.1 Attacker Model

Proving a security guarantee requires a well-defined attacker. As ℓ-reentrancy is parameterized on a label, we also parameterize attackers over what they compromise. We assume that an attacker $\mathcal{A}$ controls some collection of system components, including anything that trusts any combination of those components. For simplicity, we require a label $\ell_{\mathcal{A}}$ representing the combined attacker power and a label ℓ_t representing the minimum honest integrity, where every label is either attacker-control or honest. That is, for all $\ell \in \mathcal{L}$, either $\ell_{\mathcal{A}} \Rightarrow \ell$ or $\ell \Rightarrow \ell_t$, but not both.[2] We prove that, for *any* such ℓ_t and $\ell_{\mathcal{A}}$, if all code trusted at ℓ_t abides by the static and dynamic locking requirements, the system is ℓ-reentrancy secure whenever $\ell \Rightarrow \ell_t$. This parameterization of the attacker ensures that only someone you trust can damage your security.

Notably, the requiring ℓ_t and $\ell_{\mathcal{A}}$ to exist means that, to guaranteeing security at $\ell_1 \wedge \ell_2$, one or both of ℓ_1 and ℓ_2 must act for ℓ_t, and therefore be honest. In other words, trusting the combined power of two labels is a statement that you believe at least one of those labels is honest, though you may not know which. Combined with trust in $\ell_1 \vee \ell_2$ expressing trust in both ℓ_1 and ℓ_2, this idea supports modeling complex assumptions like "at least k of n nodes are honest."

Because reentrancy attacks stem from attacker code performing unexpected operations, we grant attackers considerable power. Specifically, attackers can modify or replace any code that executes with low integrity—that is, any code where $\ell_{\mathcal{A}} \Rightarrow pc$. Allowing attackers to modify high-integrity code executing with a low-integrity pc may seem unrealistic, but experience has shown that code bases contain "gadgets" that attackers can combine to achieve arbitrary functionality [138,

[2] Our results hold for any partition of $\mathcal{L}$ into a downward-closed sublattice $\mathcal{T}$ and an upward-closed sublattice $\mathcal{A}$, letting ℓ be "trusted" if $\ell \in \mathcal{T}$. If $\mathcal{T}$ and $\mathcal{A}$ are complete, this formulation is equivalent with $\ell_t = \bigvee \mathcal{T}$ and $\ell_{\mathcal{A}} = \bigwedge \mathcal{A}$.

152]. This expansive power conservatively models the ability to exploit such gadgets without modeling the gadgets explicitly.

To model the attacker's ability to sidestep static security features, we introduce a new term to ignore static lock labels.

$$
\begin{aligned}
e &::= \cdots \mid \textsf{ignore-locks-in}\ e \\
E &::= \cdots \mid \textsf{ignore-locks-in}\ E
\end{aligned}
$$

$$[\text{E-IgnoreLocks}]\ \frac{}{\langle \textsf{ignore-locks-in}\ v \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

$$[\text{IgnoreLocks}]\ \frac{\Sigma; \Gamma; pc; \ell'_{\mathrm{I}} \vdash e : \tau \dashv \ell'_{\mathrm{O}}}{\Sigma; \Gamma; pc; \ell_{\mathrm{I}} \vdash \textsf{ignore-locks-in}\ e : \tau \dashv \ell_{\mathrm{O}}}$$

Reasoning explicitly about ill-typed code is challenging, so the formal model requires all code to type-check, but allows low-integrity code to use this new term. Using ignore-locks-in may not appear to grant the full power of ignoring the type system. After all, the type system limits the location of method calls and state modifications based on the *pc* label, which attackers cannot modify. However, low-integrity code can only interact with high-integrity code in three ways: calling high-integrity methods, returning values to high-integrity contexts, or writing to memory that high-integrity code will later read. In each case, the operational semantics includes dynamic checks to ensure memory safety and to ensure that method calls and state modifications are only performed by sufficiently trusted code—exactly what the type system asks.

Indeed, the only constraint the type system imposes that these dynamic checks do not enforce is the static locking that ignore-locks-in is designed to avoid. Modeling well-typed high-integrity code and unknown attacker code is therefore as simple as demanding that all code type-checks and high-integrity code does not use ignore-locks-in, formalized as follows.

Definition 4.3 (Lock Compliance)**.** A class table *CT complies with locks in ℓ_t-code* if, whenever

$$CT(C) = \text{class } C[\ell_C] \text{ extends } D\,\{\overline{f:\tau_f}\ ;\ K\ ;\ \overline{M}\}$$

and $\ell_C \Rightarrow \ell_t$, then ignore-locks-in does not appear syntactically in the body of any method $m \in \overline{M}$.

Strong object-level memory isolation, like that in Ethereum, reduces the information security checks of the semantics to type-checking high-integrity code. Forcing dynamic lock checks, however, requires direct support in the system runtime. As such features are uncommon, we model a system where attackers can freely ignore dynamic locks. Specifically, we extend the operational semantics with a second rule for function calls, E-CALLATK, which enables calls to attacker-controlled code without checking dynamic label locks.

$$[\text{E-CALLATK}]\ \frac{\begin{array}{c} mbody(C,m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \ell_{\mathcal{A}} \Rightarrow pc_2 \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \text{new } C(\overline{v})] \end{array}}{\langle \text{new } C(\overline{v}).m(\overline{w}) \mid C\rangle \longrightarrow \langle \text{return}_\tau\ (e' \text{ at-pc } pc_2) \mid C[\mathcal{M}, \ell_m/\mathcal{M}]\rangle}$$

This rule is identical to E-CALL, except instead of checking dynamic locks, it checks that pc_2 is untrusted ($\ell_{\mathcal{A}} \Rightarrow pc_2$).

Interestingly, in systems that require even untrusted calls to check dynamic locks—admitting only E-CALL and not E-CALLATK—trust of $\ell_1 \wedge \ell_2$ can be safe even when neither ℓ_1 nor ℓ_2 is honest. Such systems enforce ℓ_t-reentrancy security whenever *CT* complies with locks in ℓ_t-code. There can even exist labels ℓ_1 and ℓ_2 where *CT* does not comply with locks in ℓ_1-code or ℓ_2-code, but $\ell_1 \wedge \ell_2 \Rightarrow \ell_t$, meaning $\ell_{\mathcal{A}}$ cannot be a well-defined label. The proofs in Section 4.12 consider both system and attacker models.

Attacker-provided code. In addition to having ill-typed code, attackers can tailor their attacks to the specific application. We therefore define security with respect to *any* system with the same high-integrity code. Specifically, we employ a notion of ℓ_t-equivalent code that allows an attacker to add, remove, or replace code whenever $pc \not\Rightarrow \ell_t$.

We formalize the equivalence using erasure on the code in a class table CT. Let $CT|_{\ell_t}$ denote CT, but erasing any class C with low-integrity code ($\ell_C \not\Rightarrow \ell_t$), any method m that executes with low integrity ($pc_2 \not\Rightarrow \ell_t$), and the branches of if statements executing with low integrity ($pc \not\Rightarrow \ell_t$). Two class tables are then ℓ_t-equivalent if they erase to the same thing.

$$CT \approx_\ell CT' \overset{\triangle}{\iff} CT|_{\ell_t} = CT'|_{\ell_t}$$

Attackers can also freely modify low-integrity locations in the heap, so we define ℓ_t-equivalent heaps using similar erasure. As a heap σ is a partial function from locations to value–type pairs, memory is erased to $\sigma|_{\ell_t}$ simply by erasing mappings with low-integrity types. Formally, $\sigma|_{\ell_t}(\iota) = \sigma(\iota)$ if $\sigma(\iota) = (v, t^\ell)$ with $\ell \Rightarrow \ell_t$, and it is undefined otherwise. As with code, the equivalence follows directly from this erasure:

$$\sigma \approx_\ell \sigma' \overset{\triangle}{\iff} \sigma|_{\ell_t} = \sigma'|_{\ell_t}.$$

4.5.2 Noninterference

A typical goal for security in IFC systems, including our core calculus, is *noninterference* [72], which for integrity means untrusted data should not influence trusted data at all. As we argued in Section 4.2.2, noninterference is too restrictive, and indeed, endorsement exists to violate it. However, explicit endorsement should be the *only* way to violate noninterference.

To state this, we first need a notion of a class table CT being *endorsement-free* for a label ℓ.

Definition 4.4 (Endorsement-Free)**.** CT is *ℓ-endorsement-free* if, for all classes C and methods m such that

$$\textsf{class}\ C[\ell_C]\ \textsf{extends}\ D\ \{\overline{f:\tau_f}\ ;\ K\ ;\ \overline{M}\} \in CT$$

$$\tau\ m\{pc_1 \gg pc_2; \ell_O\}(\overline{x:\tau_a})\ \{e\} \in \overline{M}$$

two properties hold. (1) Either $pc_1 \Rightarrow \ell$ or $pc_2 \not\Rightarrow \ell$, and (2) for any subexpression of e of the form $\textsf{endorse}\ v\ \textsf{from}\ \ell_1\ \textsf{to}\ \ell_2$, similarly, either $\ell_1 \Rightarrow \ell$ or $\ell_2 \not\Rightarrow \ell$.

Intuitively, this definition says that CT is ℓ-endorsement-free if CT contains no means of endorsing either control flow or data from a label that ℓ does not trust to one that it does.

This condition is sufficient to prove a strong notion of noninterference at ℓ. Because the SeRIF semantics are nondeterministic with respect to selection of location names (E-Ref), we use a modified equivalence $\simeq_\ell$ that allows renaming locations in addition to erasing low-integrity state. See Section 4.9 for the formal definition of this equivalence.

For partial functions f and f', we write $f \subseteq f'$ to mean $\mathrm{dom}(f) \subseteq \mathrm{dom}(f')$ and $f(x) = f'(x)$ wherever f is defined.

Theorem 4.1 (Noninterference)**.** *Let CT be a class table where $\Sigma \vdash CT$ ok is ℓ-endorsement-free. For any well-typed heaps σ_1 and σ_2 such that $\Sigma \subseteq \Sigma_{\sigma_i}$ and any invocation I such that $\Sigma \vdash I$ and $(I, CT, \sigma_i) \Downarrow \sigma_i'$, if $\sigma_1 \simeq_\ell \sigma_2$, then $\sigma_1' \simeq_\ell \sigma_2'$.*

Theorem 4.1 follows by complicated induction on the operational semantics, erasing untrusted values in the heap. See Section 4.11 for details.

Note also, the theorem says nothing about lock compliance, only endorsement freedom. Indeed, reentrancy locks are unnecessary to enforce noninterference.

4.5.3 Formalizing Reentrancy

Definition 4.1 in Section 4.3.1 informally defines ℓ-reentrancy as a trusted operation calling an untrusted operation, which then calls a trusted operation before returning. We also noted that the *pc* label specifies the integrity of the control flow and is therefore ideal for defining reentrancy.

Because SeRIF's semantics has no explicit call stack, it must insert at-pc tracking terms in the only places where the *pc* label of the currently-executing code can change: conditionals and method calls. The terms surround the body of the condition or method and remain until execution returns to the previous *pc* label. Nested tracking terms appear precisely when code in one conditional or method body calls a second before returning. We therefore formalize ℓ-reentrancy as three nested at-pc terms where ℓ trusts the label of the first and third, but not the second. As each condition or call may still have pending computation, we allow arbitrary evaluation contexts at each integrity level.

Definition 4.5 (ℓ-Reentrancy)**.** A statement s is *ℓ-reentrant* if, for some evaluation contexts E_0, E_1, E_2,

$$s = E_0\Big[E_1[E_2[s' \text{ at-pc } pc_3] \text{ at-pc } pc_2] \text{ at-pc } pc_1\Big]$$

where $pc_1, pc_3 \Rightarrow \ell$ but $pc_2 \not\Rightarrow \ell$.

An invocation $I = (\iota, m(\overline{v}), \ell')$ is *ℓ-reentrant in σ* if $\langle !\iota.m(\overline{v}) \mid (CT, \sigma, \ell', \cdot)\rangle \longrightarrow^* \langle s \mid C\rangle$ where s is ℓ-reentrant.

With a definition of reentrancy and a formal attacker model, we can now formalize the notion of security described in Section 4.3.2. Recall that we took "secure reentrancy" to mean that any program behavior the application exhibits with reentrancy, it can also exhibit without reentrancy. Equivalently, any state changes made by a reentrant execution must be possible using non-reentrant ones.

The properties a program *maintains* can be described using a modified Hoare logic [79]. Because high-integrity code may interact with arbitrary attacker code, we consider all possible invocations with ℓ-equivalent code. Specifically, the high-integrity component of CT maintains a property defined by a predicate pair (P, Q) if, whenever P holds on the input state, Q must hold on the output state.

Definition 4.6 (Predicate Satisfaction)**.** Given a class table CT, a heap type Σ, and state predicates P and Q, *CT satisfies (P, Q) at ℓ in Σ*, denoted $\Sigma \vDash_\ell \{P\}\ CT\ \{Q\}$, if, for any CT' such that $CT \approx_\ell CT'$, any well-typed state σ_1 where $\Sigma \subseteq \Sigma_{\sigma_1}$, and any invocation sequence $\bar{I}$ such that $\Sigma_{\sigma_1} \vdash \bar{I}$ and $(\bar{I}, CT', \sigma_1) \Downarrow \sigma_2$, then $P(\sigma_1)$ implies $Q(\sigma_2)$.

To simplify proofs, the definition requires invocations to be well-typed. The requirement does not, however, weaken the security guarantee. In a system like Ethereum without a strong type system, a high-integrity contract would need to examine its arguments to ensure they are well-typed. We assume this facility is built into the runtime.

The predicates P and Q can capture a variety of program properties. A simple example is program invariants—such as Uniswap's invariant on the product of the token balances—in which case they would be the same. Quantifying over a potentially infinite set of predicates, as the security definition does below, allows for arbitrarily complex properties. For example, requiring a specific high-integrity output state for each possible high-integrity input state would enforce noninterference. A demonstration of interference would demonstrate that one such predicate pair is not satisfied.

Our goal, however, is not to guarantee any specific properties, but to formalize the idea that reentrancy should not introduce new behavior. Definition 4.6 says nothing about reentrancy. It captures the *entire* set of possible behaviors, including the reentrant ones. Saying that a complete set of behaviors is equivalent to the non-reentrant behaviors requires a definition of non-reentrant behaviors. For that, we simply restrict our previous definition to executions that are not ℓ-reentrant.

Definition 4.7 (Single-Entry Predicate Satisfaction)**.** Given a class table CT, a heap type Σ, and state predicates P and Q, we say that CT *single-entry satisfies* (P, Q) *at* ℓ *in* Σ, denoted $\Sigma \vDash^1_\ell \{P\}\ CT\ \{Q\}$, if CT satisfies (P, Q) at ℓ in Σ when restricted to invocation sequences $\overline{I}$ that are *not* ℓ-reentrant.

These two definitions combine to specify the difference between non-reentrant program behavior and all program behavior. To compare them, note that a program satisfies predicate pair (P, Q) precisely when no behavior violates it. Thus, if reentrancy can exhibit new behaviors—the program is insecure—there should be a predicate pair that is single-entry satisfied, but not satisfied in general.

Because attackers can arbitrarily modify low-integrity state, *any* changes to low-integrity state are possible without ℓ-reentrancy. We correspondingly restrict our security notion to predicates that are unaffected by low-integrity state.

Definition 4.8 (ℓ-integrity Predicate)**.** We say a predicate P is *ℓ-integrity* if, for all pairs of states σ_1 and σ_2,

$$\sigma_1 \approx_\ell \sigma_2 \implies P(\sigma_1) \Leftrightarrow P(\sigma_2).$$

We now define ℓ-reentrancy security formally.

Definition 4.9 (Reentrancy Security (formal))**.** We say a class table CT is *ℓ-reentrancy secure in* Σ if for all pairs (P, Q) of ℓ-integrity predicates, $\Sigma \vDash^1_\ell \{P\} CT \{Q\}$ implies $\Sigma \vDash_\ell \{P\}\ CT\ \{Q\}$.

Definition 4.9 is the core security definition SeRIF enforces.

Theorem 4.2. *For any label ℓ, class table CT, and heap type Σ, if $\ell \Rightarrow \ell_t$ and $\Sigma \vdash CT$* ok *complies with locks in ℓ_t-code, then CT is ℓ-reentrancy secure in Σ.*

Theorem 4.2 follows from two core results. First, all reentrancy allowed by SeRIF is tail reentrancy. That is, if an invocation passes through an ℓ-reentrant state, then the outer high-integrity call (E_1 at-pc pc_1 in Definition 4.5) must be in tail position.

Theorem 4.3. *For a label ℓ, class table CT, and well-typed heap σ_1, if $\ell \Rightarrow \ell_t$ and $\Sigma_{\sigma_1} \vdash CT$ ok complies with locks in ℓ_t-code, then for any invocation I and heap σ_2 where $\Sigma_{\sigma_1} \vdash I$ and $(I, CT, \sigma_1) \Downarrow \sigma_2$, all ℓ-reentrant states in the execution are ℓ-tail-reentrant.*

Proof Sketch. The theorem follows from two facts. First, if a statement s steps to a call to a method that grants integrity ℓ, then s cannot maintain a lock on ℓ. Second, any statement executing with integrity ℓ must maintain a lock on ℓ (either statically or dynamically) unless it is in tail position. We provide a complete proof in Section 4.12.1. □

Once we know that all reentrant executions are tail-reentrant, we need only show that tail reentrancy is secure. The following theorem formalizes this idea by proving that, if all ℓ-reentrant states are ℓ-tail-reentrant, then single-entry predicate satisfaction translates to predicate satisfaction.

Theorem 4.4. *Let CT be a class table, σ_1 and σ_2 be well-typed heaps, and I be an invocation such that $(I, CT, \sigma_1) \Downarrow \sigma_2$ where all ℓ-reentrant states are ℓ-tail-reentrant. For any ℓ-integrity predicates P and Q, if $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$ and $P(\sigma_1)$, then $Q(\sigma_2)$.*

Proof Sketch. Examine the execution of I and build a CT' and $\overline{I}$ that produce a ℓ-equivalent final state with no reentrancy. Whenever a high-integrity environment transitions to a low-integrity one in CT, replace the low-integrity code in CT' with code that returns the same value as a hard-coded constant and makes no calls to high-integrity code. For each call from a low-integrity environment to a high-integrity method, add an invocation to $\overline{I}$ that makes the same call with the same arguments. Add additional invocations between each high-integrity call to update the low-integrity state to match the low-integrity state in the original execution when the call occurred. The result is clearly a non-reentrant set of executions. Because all ℓ-reentrant states are ℓ-tail-reentrant in the original execution, placing a reentrant call sequentially after the call it was originally inside produces the same result.

Since the start and end states σ_1' and σ_2' of this new execution are ℓ-equivalent to σ_1 and σ_2 and $\Sigma_{\sigma_1} \vDash_\ell^1 \{P\}\ CT\ \{Q\}$,

$$P(\sigma_1) \iff P(\sigma_1') \implies Q(\sigma_2') \iff Q(\sigma_2).$$

See Section 4.12.2 for details. □

From here, we have enough to prove our desired result.

Proof of Theorem 4.2. For a class table CT', invocation I, and heaps σ_1 and σ_2 such that $CT \approx_\ell CT'$ and $(I, CT', \sigma_1) \Downarrow \sigma_2$, Theorem 4.3 says all ℓ-reentrant states are ℓ-tail-reentrant. For ℓ-integrity predicates P and Q such that $\Sigma_{\sigma_1} \vDash_\ell^1 \{P\}\ CT\ \{Q\}$, Theorem 4.4 says that if $P(\sigma_1)$ then $Q(\sigma_2)$, which is precisely the definition of $\Sigma_{\sigma_1} \vDash_\ell \{P\}\ CT\ \{Q\}$. □

4.6 Implementation

We implemented a type checker for SeRIF in 4,200 lines of Java, using JFlex [89] and CUP [81]. We employ the SHErrLoc constraint solver [181] to analyze information flow constraints, infer missing integrity labels, and identify likely error locations.

We ran the type checker on four examples: the three from Section 4.1, but without simplifying Town Crier, and one we call Multi-DAO. Multi-DAO is a multi-contract version of the vulnerable portion of Ethereum's DAO contract [131]. It is one application split across multiple contracts that synchronize on each transaction. This structure allows for the DAO's original reentrancy vulnerability, as well as a second attack where the attacker reenters the application by leaving one contract and entering another before they synchronize. By definition, this attack is not object reentrancy, but as long as the Multi-DAO contracts trust each other, it *is* ℓ-reentrancy. As with the original DAO, the exploits can be patched either with dynamic locks or by performing local state changes *and inter-contract synchronization operations* before external calls.

For each example the type checker correctly identified vulnerabilities in the initial versions presented in Section 4.1. It also accepted as secure patched implementations following the suggested fixes, both with and without dynamic locks.

Developer Overhead. Table 4.1 presents several metrics for developer overhead. As each example application is designed to distill complex security logic into minimal code, the examples are all relatively short—ranging from 35 to 133 lines of code. On these examples, the type checker is able to run in a few seconds on a consumer desktop from 2015 with an Intel i7-4790 CPU. Because the type system and the associated guarantees are compositional, modules can be checked independently, so running time should scale well as the code grows.

Another important practical concern is the annotation burden of adding information flow labels to the code. Labels on classes, fields, methods, and data endorsements are necessary to define the security of a program. Though SeRIF requires explicit labels elsewhere to ease formal reasoning, many of these—such as the *pc* labels on `if` statements—are simple to infer. Considering only the labels with no obvious inference mechanism, we found that 13% of the lines required explicit labels in Town Crier. The other examples required more annotations per line as their distilled nature led to more function declarations and explicit endorsements.

Application	LoC	type-check time (s)	necessary annotations
Uniswap 1	57	4.1	11
Uniswap 2	49	4.0	9
Uniswap 3*	53	4.3	9
Town Crier 1	133	6.3	17
Town Crier 2*	133	6.5	17
Town Crier 3*	133	6.4	17
KV Store 1	38	2.1	10
KV Store 2*	35	2.0	9
Multi-DAO 1	38	3.5	8
Multi-DAO 2	36	3.3	7
Multi-DAO 3*	36	3.3	7

Table 4.1: Evaluation of SeRIF type checker. Asterisks indicate vulnerable implementations.

As even Town Crier is a short application with complex security concerns, we expect many applications would have lower annotation burdens.

Finally, SHErrLoc is capable of localizing errors, helping guide development. To see its utility, we look at the Uniswap example in more detail. As in Section 4.4.4, we use two labels: U and T. Recall that the exchange must either utilize a lock or state its assumption that the token will not call untrusted code. The following signature for the token's `transferTo` method makes the assumption explicit, where `H` is a token holder class.

$$\texttt{bool}^{T}\ \texttt{transferTo}\{T \gg T; T\}(\texttt{from:H}^{T},\ \texttt{to:H}^{T},\ \texttt{amount:int}^{T})$$

To model the alert functions in `H` being unknown code from unknown sources, the interface can state the following entirely-untrusted signatures.

$$\texttt{void alertSend}\{U \gg U; U\}(\texttt{to:H}^{U},\ \texttt{amount:int}^{U})$$
$$\texttt{void alertReceive}\{U \gg U; U\}(\texttt{from:H}^{U},\ \texttt{amount:int}^{U})$$

With these signatures, the calls to the alert functions in `transferTo` on lines 20 and 21 of Figure 4.1 cannot type-check without a dynamic lock. SHErrLoc helpfully identifies line 21 as the most likely error. The type checker correctly identifies the program as secure if we either wrap both alerts in a dynamic lock or remove them entirely.

4.7 Related Work

We now discuss other work on reentrancy security, secure smart contracts, and information flow control.

Formal Reentrancy Security. Grossman et al. [75] define Effectively Callback-Free (ECF) executions, the only other formal definition of reentrancy security of which we are aware. An ECF execution is one where the operations can be re-ordered to produce the same result without callbacks (reentrancy). Their definition is object-based, which we have argued fails to separate the security specification from the program design, and they focus on dynamic analysis of individual executions.

Albert et al. [5] present a static analysis tool to check if code produces only ECF executions. The authors advertise the tool as providing modular guarantees, but define "modular" to mean that a contract remains secure against any possible outside code. Our approach provides the same guarantees when applied to a single program with no assumptions on others, but also enables developers to safely compose independently-checked modules by stating assumptions on each other's behavior. Furthermore, Albert et al.'s analysis relies on an SMT solver, limiting its scalability. In comparison, SeRIF only relies on checking acts-for relationships of information flow labels.

We previously proposed the intuition of using information flow control with a mix of static and dynamic locks to enforce ℓ-reentrancy [38]. In this work we add a core calculus with static and dynamic semantics, formal definitions, proofs, and an evaluation.

Reentrancy-aware Languages. Several languages—all smart-contract oriented—attempt to guard against reentrancy using a variety of techniques.

Scilla [150] constrains programming style by eliminating the call-and-return model of contract interaction. Instead, it queues requests and executes them when the caller completes. While this structure makes object-level reentrancy difficult, it prevents contracts from using the return values from remote calls. Moreover, by allowing multiple unconstrained requests, it fails to detect or eliminate bugs like Uniswap (see Section 4.1.1).

Obsidian [47] and Flint [149] ease reasoning about contract behavior using typestate. Obsidian includes a dynamic check that prevents (object) reentrancy entirely, while Flint has no such check. Both languages and Move [20] have a notion of linear assets that cannot be created or destroyed. Asset linearity prevents attacks like the DAO, but fails to address the challenges of Uniswap. The errant send in Uniswap does not create or destroy tokens; it merely sends the wrong number because it the invariant it relies on is broken.

Nomos [57] enforces security using resource-aware session types. Since linearity of session types is insufficient to eliminate reentrancy, It uses the resources tracked by the session types to prevent attackers from acquiring permission to call an in-use contract—again, eliminating all (object) reentrancy.

Smart Contract Analysis Tools. There are many static analysis tools for blockchain smart contracts. Some tools operate as best-effort bug finding tools with no soundness guarantees. OYENTE [100] searches for anti-patterns in code, TEETHER [93] automatically generates exploits based on commonly-exploitable operations, and Ethainter [26] uses information flow taint analysis to attempt to locate a predefined set of security concerns, such as tainted owner variables and access to self-destruct.

Other tools use formal analysis techniques to soundly analyze contracts. Bhargavan et al. [18] prove functional correctness by translation to F*. MAIAN [123] and ETHBMC [68] prove security against specific classes of vulnerabilities using symbolic execution and bounded model checking, respectively. EtherTrust [74] allows developers to specify program properties as Horn clauses and verify them using a formal semantics for EVM [73]. SOLYTHESIS [95] combines static and dynamic mechanisms It statically determines what checks are necessary for correctness and compiles them into run-time checks.

These tools are valuable for securing smart contracts, but they all analyze individual contracts, and their analyses often fail to compose. As a result, they are unable to verify security of applications like Uniswap that span multiple contracts.

Information Flow Control. Several distributed and decentralized systems enforce security using IFC. Fabric [98] is a system and language for building distributed systems that allows secure data and code sharing between nodes despite mutual distrust. DStar [179] uses run-time tracking at the OS level to control information flow in a distributed system. These previous systems have the same limitation of information flow systems that is described in the introduction to this chapter and Section 4.2.2: they do not defend against reentrancy attacks. The IFC-based instruction set of Zagieboylo et al. [176] restricts endorsement of *pc* labels using a purely dynamic mechanism that appears to prevent all ℓ-reentrancy. However, this property is neither stated nor proved.

4.8 Full SeRIF Rules

The full operational semantics for SeRIF is given in Figure 4.9 and the full typing rules are given in Figure 4.10.

[E-Eval]
$$\frac{\langle s \mid C\rangle \longrightarrow \langle s' \mid C'\rangle}{\langle E[s] \mid C\rangle \longrightarrow \langle E[s'] \mid C'\rangle}$$

[E-Let]
$$\frac{}{\langle \textsf{let}\ x = v\ \textsf{in}\ e \mid C\rangle \longrightarrow \langle e[x \mapsto v] \mid C\rangle}$$

[E-IfT]
$$\frac{}{\langle \textsf{if}\{pc\}\ \textsf{true then}\ e_1\ \textsf{else}\ e_2 \mid C\rangle \longrightarrow \langle e_1\ \textsf{at-pc}\ pc \mid C\rangle}$$

[E-IfF]
$$\frac{}{\langle \textsf{if}\{pc\}\ \textsf{false then}\ e_1\ \textsf{else}\ e_2 \mid C\rangle \longrightarrow \langle e_2\ \textsf{at-pc}\ pc \mid C\rangle}$$

[E-AtPc]
$$\frac{}{\langle v\ \textsf{at-pc}\ pc \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

[E-Ref]
$$\frac{\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \textsf{ref}\ v\ \tau \mid C\rangle \longrightarrow \langle \iota \mid C[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

[E-Deref]
$$\frac{\sigma(\iota) = (v, \tau)}{\langle !\iota \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

[E-Assign]
$$\frac{\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \triangleleft \tau}{\langle \iota := v \mid C\rangle \longrightarrow \langle () \mid C[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

[E-Cast]
$$\frac{D <: C}{\langle (C)(\textsf{new}\ D(\overline{v})) \mid C\rangle \longrightarrow \langle \textsf{new}\ D(\overline{v}) \mid C\rangle}$$

[E-Field]
$$\frac{}{\langle \textsf{new}\ C(\overline{v}).f_i \mid C\rangle \longrightarrow \langle v_i \mid C\rangle}$$

[E-Endorse]
$$\frac{}{\langle \textsf{endorse}\ v\ \textsf{from}\ \ell'\ \textsf{to}\ \ell \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

(a) Standard IFC calculus small-step semantic rules

$$[\text{E-CALL}]\quad \frac{\begin{array}{c} mbody(C,m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \bigwedge_{\ell \in L} (pc_1 \Rightarrow pc_2 \vee \ell) \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \textsf{this} \mapsto \textsf{new}\ C(\overline{v})] \end{array}}{\langle \textsf{new}\ C(\overline{v}).m(\overline{w}) \mid C\rangle \longrightarrow \langle \textsf{return}_\tau\ (e'\ \textsf{at-pc}\ pc_2) \mid C[\mathcal{M}, \ell_m/\mathcal{M}]\rangle}$$

$$[\text{E-CALLATK}]\quad \frac{\begin{array}{c} mbody(C,m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau) \\ \mathcal{M} = \mathcal{M}', \ell'_m \qquad \ell'_m \Rightarrow pc_1 \qquad \ell_{\mathcal{A}} \Rightarrow pc_2 \\ \Sigma_\sigma \vdash \overline{w} : \overline{\tau_a} \qquad e' = e[\overline{x} \mapsto \overline{w}, \textsf{this} \mapsto \textsf{new}\ C(\overline{v})] \end{array}}{\langle \textsf{new}\ C(\overline{v}).m(\overline{w}) \mid C\rangle \longrightarrow \langle \textsf{return}_\tau\ (e'\ \textsf{at-pc}\ pc_2) \mid C[\mathcal{M}, \ell_m/\mathcal{M}]\rangle}$$

$$[\text{E-RETURN}]\quad \frac{\Sigma_\sigma \vdash v : \tau \qquad \mathcal{M} = \mathcal{M}', \ell_m}{\langle \textsf{return}_\tau\ v \mid C\rangle \longrightarrow \langle v \mid C[\mathcal{M}'/\mathcal{M}]\rangle}$$

$$[\text{E-LOCK}]\quad \frac{}{\langle \textsf{lock}\ \ell\ \textsf{in}\ e \mid C\rangle \longrightarrow \langle e\ \textsf{with-lock}\ \ell \mid C[L, \ell/L]\rangle}$$

$$[\text{E-UNLOCK}]\quad \frac{L = L', \ell}{\langle v\ \textsf{with-lock}\ \ell \mid C\rangle \longrightarrow \langle v \mid C[L'/L]\rangle}$$

$$[\text{E-IGNORELOCKS}]\quad \frac{}{\langle \textsf{ignore-locks-in}\ v \mid C\rangle \longrightarrow \langle v \mid C\rangle}$$

(b) Lock-aware small-step semantic rules

Figure 4.9: Full small-step operational semantics for SeRIF.

$$[\text{VAR}]\ \frac{\Gamma(x) = \tau}{\Sigma; \Gamma \vdash x : \tau} \qquad [\text{UNIT}]\ \frac{}{\Sigma; \Gamma \vdash () : \textsf{unit}^\ell} \qquad [\text{TRUE}]\ \frac{}{\Sigma; \Gamma \vdash \textsf{true} : \textsf{bool}^\ell}$$

$$[\text{FALSE}]\ \frac{}{\Sigma; \Gamma \vdash \textsf{false} : \textsf{bool}^\ell} \qquad [\text{NEW}]\ \frac{\begin{array}{c} fields(C) = \overline{f} : \overline{\tau} \\ \Sigma; \Gamma \vdash \overline{v} : \overline{\tau} \end{array}}{\Sigma; \Gamma \vdash \textsf{new}\ C(\overline{v}) : C^\ell} \qquad [\text{LOC}]\ \frac{\Sigma(\iota) = \tau}{\Sigma; \Gamma \vdash \iota : (\textsf{ref}\ \tau)^\ell}$$

$$[\text{NULL}]\ \frac{}{\Sigma; \Gamma \vdash \textsf{null} : (\textsf{ref}\ \tau)^\ell} \qquad [\text{SUBTYPEV}]\ \frac{\Sigma; \Gamma \vdash v : \tau' \qquad \tau' <: \tau}{\Sigma; \Gamma \vdash v : \tau}$$

(a) Value typing

$$[\textsc{Val}]\ \frac{\Sigma;\Gamma\vdash v:\tau}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash v:\tau\dashv\ell_{\mathrm{O}}}\qquad [\textsc{Endorse}]\ \frac{\Sigma;\Gamma\vdash v:t^{\ell'}}{\Sigma;\Gamma;\ell;\ell_{\mathrm{I}}\vdash \textsf{endorse}\ v\ \textsf{from}\ \ell'\ \textsf{to}\ \ell : t^{\ell}\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Cast}]\ \frac{\Sigma;\Gamma\vdash v:D^{\ell}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash (C)v:C^{\ell}\dashv\ell_{\mathrm{O}}}\qquad [\textsc{Field}]\ \frac{\begin{array}{c}\Sigma;\Gamma\vdash v:C^{\ell}\qquad \mathit{fields}(C)=\overline{f}:\overline{\tau}\\ \tau_i <: \tau\qquad \ell\triangleleft\tau\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash v.f_i:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c}\mathit{mtype}(C,m)=\overline{\tau_a}\xrightarrow{pc_1\gg pc_2;\ell_{\mathrm{O}}}\tau_0\qquad \Sigma;\Gamma\vdash v:C^{\ell}\qquad \Sigma;\Gamma\vdash\overline{v_a}:\overline{\tau_a}\\ \ell\Rightarrow pc_1\qquad pc_1\Rightarrow pc_2\vee\ell_{\mathrm{I}}\qquad \tau_0<:\tau\qquad pc_2\vee\ell\triangleleft\tau\end{array}}{\Sigma;\Gamma;pc_1;\ell_{\mathrm{I}}\vdash v.m(\overline{v_a}):\tau\dashv\ell_{\mathrm{O}}\vee pc_2}$$

$$[\textsc{If}]\ \frac{\begin{array}{c}\Sigma;\Gamma\vdash v:\textsf{bool}^{\ell}\qquad \ell\Rightarrow pc\qquad \ell\triangleleft\tau\\ \Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash e_1:\tau\dashv\ell_{\mathrm{O}}\qquad \Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash e_2:\tau\dashv\ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{if}\{pc\}\ v\ \textsf{then}\ e_1\ \textsf{else}\ e_2:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Ref}]\ \frac{\Sigma;\Gamma\vdash v:\tau\qquad pc\triangleleft\tau}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{ref}\ v\ \tau:(\textsf{ref}\ \tau)^{\ell}\dashv\ell_{\mathrm{O}}}\qquad [\textsc{Deref}]\ \frac{\begin{array}{c}\Sigma;\Gamma\vdash v:(\textsf{ref}\ \tau')^{\ell}\\ \tau'<:\tau\qquad \ell\triangleleft\tau\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash\ !v:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Assign}]\ \frac{\begin{array}{c}\Sigma;\Gamma\vdash v_1:(\textsf{ref}\ \tau)^{\ell}\\ \Sigma;\Gamma\vdash v_2:\tau\qquad \ell\triangleleft\tau\end{array}}{\Sigma;\Gamma;\ell;\ell_{\mathrm{I}}\vdash v_1:=v_2:\textsf{unit}^{\ell'}\dashv\ell_{\mathrm{O}}}\qquad [\textsc{Lock}]\ \frac{\begin{array}{c}\Sigma;\Gamma;pc;\ell'_{\mathrm{I}}\vdash e:\tau\dashv\ell'_{\mathrm{O}}\\ \ell'_{\mathrm{I}}\wedge\ell\Rightarrow\ell_{\mathrm{I}}\qquad \ell'_{\mathrm{O}}\wedge\ell\Rightarrow\ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{lock}\ \ell\ \textsf{in}\ e:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Let}]\ \frac{\begin{array}{c}\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash e_1:\tau_1\dashv\ell'_{\mathrm{O}}\qquad \ell'_{\mathrm{O}}\Rightarrow\ell_{\mathrm{I}}\\ \Sigma;\Gamma,x{:}\tau_1;pc;\ell_{\mathrm{I}}\vdash e_2:\tau_2\dashv\ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{let}\ x=e_1\ \textsf{in}\ e_2:\tau_2\dashv\ell_{\mathrm{O}}}\qquad [\textsc{Variance}]\ \frac{\begin{array}{c}\Sigma;\Gamma;pc';\ell'_{\mathrm{I}}\vdash e:\tau'\dashv\ell'_{\mathrm{O}}\\ \tau'<:\tau\qquad pc\Rightarrow pc'\\ \ell'_{\mathrm{I}}\Rightarrow\ell_{\mathrm{I}}\qquad \ell'_{\mathrm{O}}\Rightarrow\ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash e:\tau\dashv\ell_{\mathrm{O}}}$$

(b) Core expression typing

$$[\textsc{AtPc}]\ \frac{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash s:\tau\dashv\ell_{\mathrm{O}}}{\Sigma;\Gamma;pc';\ell_{\mathrm{I}}\vdash s\ \textsf{at-pc}\ pc:\tau\dashv\ell_{\mathrm{O}}}\qquad [\textsc{WithLock}]\ \frac{\begin{array}{c}\Sigma;\Gamma;pc;\ell'_{\mathrm{I}}\vdash s:\tau\dashv\ell'_{\mathrm{O}}\\ \ell'_{\mathrm{I}}\wedge\ell\Rightarrow\ell_{\mathrm{I}}\qquad \ell'_{\mathrm{O}}\wedge\ell\Rightarrow\ell_{\mathrm{O}}\end{array}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash s\ \textsf{with-lock}\ \ell:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{Return}]\ \frac{\Sigma;\cdot;pc;\ell'_{\mathrm{I}}\vdash s:\tau\dashv\ell'_{\mathrm{O}}\qquad \ell'_{\mathrm{I}}\vee\ell'_{\mathrm{O}}\Rightarrow\ell_{\mathrm{O}}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{return}_{\tau}\ s:\tau\dashv\ell_{\mathrm{O}}}$$

$$[\textsc{IgnoreLocks}]\ \frac{\Sigma;\Gamma;pc;\ell'_{\mathrm{I}}\vdash e:\tau\dashv\ell'_{\mathrm{O}}}{\Sigma;\Gamma;pc;\ell_{\mathrm{I}}\vdash \textsf{ignore-locks-in}\ e:\tau\dashv\ell_{\mathrm{O}}}$$

(c) Tracking and attacker-modeling statement typing

$$\text{[Method-Ok]}\ \frac{\begin{array}{c}\ell_I \Rightarrow pc_2 \qquad \ell_C \Rightarrow pc_2 \qquad \ell_I \vee \ell'_O \Rightarrow \ell_O \qquad pc_1 \triangleleft \overline{\tau_a} \\ \Sigma; \overline{x}:\overline{\tau_a}, \mathsf{this}: C^{pc_2}; pc_2; \ell_I \vdash e : \tau \dashv \ell'_O \\ CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\} \\ (D,m) \in \mathrm{dom}(mtype) \implies mtype(D,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \tau\end{array}}{\Sigma \vdash \tau\ m\{pc_1 \gg pc_2; \ell_O\}(\overline{x}:\overline{\tau_a})\ \{e\}\ \mathsf{ok\ in}\ C}$$

$$\text{[Class-Ok]}\ \frac{\begin{array}{c} \mathit{fields}(D) = \overline{g}:\overline{\tau_g} \\ K = C(\overline{g}:\overline{\tau_g}\ ;\ \overline{f}:\overline{\tau_f})\ \{\mathsf{super}(\overline{g})\ ;\ \mathsf{this}.\overline{f} = \overline{f}\} \\ \Sigma \vdash \overline{M}\ \mathsf{ok\ in}\ C\end{array}}{\Sigma \vdash \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}:\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\}\ \mathsf{ok}}$$

$$\text{[CT-Ok]}\ \frac{\begin{array}{c} C \text{ referenced in any type} \implies C \in \mathrm{dom}(CT) \\ \forall C \in \mathrm{dom}(CT).\, \Sigma \vdash CT(C)\ \mathsf{ok}\end{array}}{\Sigma \vdash CT\ \mathsf{ok}}$$

(d) Class typing

$$\frac{\begin{array}{c} CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}:\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ \mathit{fields}(D) = \overline{g}:\overline{\tau_g}\end{array}}{\mathit{fields}(C) = \overline{g}:\overline{\tau_g}\ ;\ \overline{f}:\overline{\tau_f}}$$

$$\frac{\begin{array}{c} CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}:\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ \tau\ m\{pc_1 \gg pc_2; \ell_O\}(\overline{x}:\overline{\tau_a})\ \{e\} \in \overline{M}\end{array}}{\begin{array}{c} mtype(C,m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \tau \\ mbody(C,m) = (\ell_C, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)\end{array}}$$

$$\frac{\begin{array}{c} CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\overline{f}:\overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ m \text{ not defined in } \overline{M}\end{array}}{\begin{array}{c} mtype(C,m) = mtype(D,m) \\ mbody(C,m) = mbody(D,m)\end{array}}$$

(e) Lookup functions

$$\frac{\ell \Rightarrow \ell'}{t^{\ell} <: t^{\ell'}} \qquad \frac{CT(C) = \mathsf{class}\ C[\ell_C]\ \mathsf{extends}\ D\ \{\cdots\}}{C^{\ell} <: D^{\ell}} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

(f) Subtyping rules

$$\frac{\ell \Rightarrow \ell'}{\ell \triangleleft t^{\ell'}}$$

(g) Protection relation

$$\frac{\sigma(\iota) = (v, \tau) \implies \Sigma_\sigma \vdash v : \tau}{\vdash \sigma \ \mathsf{wt}}$$

(h) Heap typing

Figure 4.10: Full typing rules for SeRIF.

4.9 Location–Name Isomorphism

The E-REF operational semantic rule allows for selection of any unmapped location name when creating a new location. This makes the SeRIF operational semantics nondeterministic in its choice of location names. However, this is the only source of nondeterminism in the semantics. That is, for any pair of statement-heap pairs that are equivalent up to location names, if one steps, then the other steps and the results are again equivalent up to location names.

To reason about these differences, we define an equivalence relation that relates statements and heaps that differ only in their location names. Formally, we define a location name permutation θ as an injective map from locations to locations. We extend it to values by permuting location names, recursively permuting constructor arguments of objects, and leaving other values unmodified. We further extend it to statements by recursively applying to each sub-statement and to heaps as follows.

$$\theta(\sigma)(\iota) \triangleq (\theta(v), \tau) \ \text{ where } \ \sigma(\theta^{-1}(\iota)) = (v, \tau)$$

This permutation supports the requisite equivalence relation.

Definition 4.10 (Location–name isomorphism)**.** Statements s_1 and s_2 are *location–name isomorphic*, denoted $s_1 \simeq s_2$, if there exists some θ such that $s_1 = \theta(s_2)$. Similarly, for heaps σ_1 and σ_2, $\sigma_1 \simeq \sigma_2 \overset{\triangle}{\iff} \exists\theta.\, \sigma_1 = \theta(\sigma_2)$.

We write $(s_1, \sigma_1) \simeq (s_2, \sigma_2)$ to mean there is a θ such that $(s_1, \sigma_1) = (\theta(s_2), \theta(\sigma_2))$ and similarly for $(s_1, C_1) \simeq (s_2, C_2)$.

This definition is sufficient to state and prove the important property that the SeRIF semantics is deterministic up to location–name isomorphism.

Theorem 4.5. *For any s_1, s_1', and s_2 and any C_1, C_1' and C_2, if $(s_1, C_1) \simeq (s_2, C_2)$ and $\langle s_1 \mid C_1 \rangle \longrightarrow \langle s_1' \mid C_1' \rangle$, then there exists s_2' and C_2' such that $\langle s_2 \mid C_2 \rangle \longrightarrow \langle s_2' \mid C_2' \rangle$, and for all such s_2' and C_2', $(s_1', C_1') \simeq (s_2', C_2')$.*

Proof. By induction on the operational semantics. We take the permutation to be defined only mapping location names between σ_1 and σ_2 and extend it on uses of E-REF (or inductively with E-EVAL). □

For the noninterference theorem (Theorem 4.1), we combine location–name isomorphism with ℓ_t-equivalence.

Definition 4.11 (Location–name ℓ_t-isomorphism)**.** Two states σ_1 and σ_2 are *location–name ℓ_t-isomorphic*, denoted $\sigma_1 \simeq_\ell \sigma_2$, if there exists a θ such that $\sigma_1|_{\ell_t} = \theta(\sigma_2)|_{\ell_t}$.

4.10 Preservation and Progress

We now prove preservation and progress theorems for SeRIF.

Because SeRIF is stateful, the type preservation theorem includes preservation of both the statement and the heap.

Theorem 4.6 (Type Preservation)**.** *If*

- $\langle s \mid (CT, \sigma, \mathcal{M}, L) \rangle \longrightarrow \langle s' \mid (CT, \sigma', \mathcal{M}', L') \rangle$,
- $\Sigma_\sigma \vdash CT$ ok,
- $\Sigma_\sigma; \Gamma; pc; \ell_\mathrm{I} \vdash s : \tau \dashv \ell_\mathrm{O}$, *and*
- $\vdash \sigma$ wt,

then

- $\Sigma_\sigma \subseteq \Sigma_{\sigma'}$,

- $\vdash \sigma'$ wt, *and*
- $\Sigma_{\sigma'}; \Gamma; pc; \ell_I \vdash s' : \tau \dashv \ell_O$.

The proof of Theorem 4.6 makes use of several simple lemmas.

Lemma 4.1 (Closed Value Typing)**.** *If $v \neq x$ and $\Sigma; \Gamma \vdash v : t^{\ell}$, then $\Sigma; \Gamma' \vdash v : t^{\ell'}$ for any Γ' and ℓ'.*

Proof. By inspection on the value typing rules. □

Lemma 4.2 (Value Substitution)**.** *The following rule is admissible*

$$\frac{\Sigma; \Gamma, x{:}\tau'; pc; \ell_I \vdash s : \tau \dashv \ell_O \qquad \Sigma; \Gamma \vdash v : \tau'}{\Sigma; \Gamma; pc; \ell_I \vdash s[x \mapsto v] : \tau \dashv \ell_O}$$

Proof. By simple structural induction on the proof that $\Sigma; \Gamma, x{:}\tau'; pc; \ell_I \vdash s : \tau \dashv \ell_O$. □

Lemma 4.3 (Heap-type Extension)**.** *The following rules are admissible*

$$\frac{\Sigma; \Gamma \vdash v : \tau \qquad \Sigma \subseteq \Sigma'}{\Sigma'; \Gamma \vdash v : \tau} \qquad\qquad \frac{\Sigma; \Gamma; pc; \ell_I \vdash s : \tau \dashv \ell_O \qquad \Sigma \subseteq \Sigma'}{\Sigma'; \Gamma; pc; \ell_I \vdash s : \tau \dashv \ell_O}$$

Proof. By simple induction on the proofs of $\Sigma; \Gamma \vdash v : \tau$ and $\Sigma; \Gamma; pc; \ell_I \vdash s : \tau \dashv \ell_O$. □

Lemma 4.4 (Heap Extension)**.** *The following rule is admissible*

$$\frac{\vdash \sigma \text{ wt} \qquad \Sigma_\sigma \vdash v : \tau \qquad \iota \notin \text{dom}(\sigma)}{\vdash \sigma[\iota \mapsto (v, \tau)] \text{ wt}}$$

Proof. For notational ease, let $\sigma[\iota \mapsto (v, \tau)] = \sigma'$. First note that, because $\iota \notin \text{dom}(\sigma)$, it must be the case that $\Sigma_{\sigma'} = \Sigma_\sigma \cup \{\iota \mapsto \tau\}$ with $\iota \notin \text{dom}(\Sigma_\sigma) = \text{dom}(\sigma)$. Now assume $\sigma'(\iota') = (v', \tau')$. If $\iota' = \iota$, then the premise of the rule gives us $\Sigma_\sigma \vdash v' : \tau'$, otherwise inversion on $\vdash \sigma$ wt gives us the same property. By Lemma 4.3, $\Sigma_{\sigma'} \vdash v' : \tau'$, thereby proving $\vdash \sigma'$ wt. □

Lemma 4.5 (Statement Substitution). *If* $\Sigma; \Gamma; pc; \ell_{\mathrm{I}} \vdash E[s_1] : \tau \dashv \ell_{\mathrm{O}}$ *then there is some* Γ', pc', ℓ'_{I}, τ', *and* ℓ'_{O} *such that* $\Sigma; \Gamma'; pc'; \ell'_{\mathrm{I}} \vdash s_1 : \tau' \dashv \ell'_{\mathrm{O}}$ *and for any statement* s_2 *and heap-type* $\Sigma' \supseteq \Sigma$*, such that* $\Sigma'; \Gamma'; pc'; \ell'_{\mathrm{I}} \vdash s_2 : \tau' \dashv \ell'_{\mathrm{O}}$*, then* $\Sigma'; \Gamma; pc; \ell_{\mathrm{I}} \vdash E[s_2] : \tau \dashv \ell_{\mathrm{O}}$*.*

Proof. By simple induction on the proof of $\Sigma; \Gamma; pc; \ell_{\mathrm{I}} \vdash E[s_1] : \tau \dashv \ell_{\mathrm{O}}$. □

These lemmas are sufficient to prove type preservation.

Proof of Theorem 4.6. This will be a proof by induction on the typing rules and inversion on the operational semantics.

Case VAL**:** Values cannot step, so this is impossible.

Case ENDORSE**:** Because v must be a closed value, it type-checks with any label, so VAL proves the result.

Case CAST**:** Inversion on the operational semantics requires that $v = \mathsf{new}\ C'(\overline{v})$ and $C' <: C$. Therefore NEW, SUBTYPEV, and VAL prove the case.

Case FIELD**:** Inversion on the operational semantics says $v = \mathsf{new}\ D(\overline{v})$ and the premise of FIELD requires $\Sigma; \Gamma \vdash \mathsf{new}\ D(\overline{v}) : C^{\ell}$. By inversion on the value typing rules, $D^{\ell} <: C^{\ell}$ and $\Sigma; \Gamma \vdash v_i : \tau_i$. Therefore, SUBTYPEV is sufficient to prove $\Sigma; \Gamma \vdash v_i : \tau$, and VAL competes the case.

Case CALL**:** Inversion on the operational semantics says $v = \mathsf{new}\ D(\overline{v})$, and the premise of CALL requires $\Sigma; \Gamma \vdash \mathsf{new}\ D(\overline{v}) : C^{\ell}$. By inversion on the value typing rules, $D^{\ell} <: C^{\ell}$. By the restrictions on overriding and the fact that $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_{\mathrm{O}}} \tau$, we know that $mbody(D, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)$. METHOD-OK further requires $\Sigma; \overline{x}{:}\overline{\tau_a}, \mathsf{this}{:}\tilde{D}^{pc_2}; pc_2; \ell'_{\mathrm{I}} \vdash e : \tau \dashv \ell'_{\mathrm{O}}$ where $\ell'_{\mathrm{I}} \vee \ell'_{\mathrm{O}} \Rightarrow \ell_{\mathrm{O}}$ and $D <: \tilde{D}$. The premise that $\Sigma; \Gamma \vdash \overline{w} : \overline{\tau_a}$ and Lemma 4.2, are sufficient to prove $\Sigma; \cdot; pc_2; \ell'_{\mathrm{I}} \vdash e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ D(\overline{v})] : \tau \dashv \ell'_{\mathrm{O}}$. This result coupled with RETURN and ATPC prove that s' is well-typed.

Case IF**:** Inversion on the operational semantics requires that the step must be E-IFT or E-IFF. The appropriate premise of IF requiring the branches to type-check in the same environment and ATPC prove the case.

Case REF**:** By construction $\Sigma_{\sigma'}(\iota) = \tau$, so LOC and VAL prove the well-typed condition. Lemma 4.4 ensures $\vdash \sigma'$ wt, and $\sigma \subset \sigma'$, so $\Sigma_\sigma \subset \Sigma_{\sigma'}$.

Case DEREF**:** Inversion on the operational semantics shows the step uses E-DEREF, meaning $v = \iota$ and $\sigma(\iota) = (v', \tau)$. The assumption that $\vdash \sigma$ wt means $\Sigma_\sigma \vdash v' : \tau$, so that coupled with SUBTYPEV and VAL proves the case.

Case ASSIGN**:** Inversion on the operational semantics proves the step is E-ASSIGN, meaning $v_1 = \iota$. Inversion on the premise $\Sigma;\Gamma \vdash v_1 : (\mathsf{ref}\ \tau)^\ell$ then proves $\sigma(\iota) = (v, \tau)$, so E-ASSIGN requires $\Sigma_\sigma \vdash v_2 : \tau$. The heap type is therefore unchanged—$\Sigma_\sigma = \Sigma_{\sigma'}$—and $\sigma' = \sigma[\iota \mapsto (v_2, \tau)]$ remains well-typed. Finally, UNIT and VAL prove s' properly type-checks.

Case LOCK**:** The semantic rule must be E-LOCK, so $s' = e$ with-lock ℓ, and the premises of WITHLOCK are identical to LOCK, so WITHLOCK proves the case.

Case LET**:** Here we see $s = (\mathsf{let}\ x = s_1\ \mathsf{in}\ e_2)$. We consider two sub-cases: if $s_1 = v$ is a value, and if it is not. In the first sub-case, the operational semantic rule must be LET, and inversion on the typing rules proves that $\Sigma_\sigma;\Gamma \vdash v : \tau_1$, so Lemma 4.2 proves the sub-case.

In the second sub-case, inversion on the operational semantics proves that the step must be E-EVAL. The LET rule's first premise is: $\Sigma_\sigma;\Gamma;pc;\ell_\mathrm{I} \vdash s_1 : \tau_1 \dashv \ell'_\mathrm{O}$. The premise of E-EVAL is $\langle s_1 \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow \langle s'_1 \mid (CT, \sigma', \mathcal{M}', L')\rangle$, so the inductive hypothesis proves that $\Sigma_{\sigma'};\Gamma;pc;\ell_\mathrm{I} \vdash s'_1 : \tau_1 \dashv \ell'_\mathrm{O}$ with $\Sigma_\sigma \subseteq \Sigma_{\sigma'}$ and $\vdash \sigma'$ wt. Lemma 4.3 shows that $\Sigma_{\sigma'};\Gamma, x{:}\tau_1;pc;\ell_\mathrm{I} \vdash e_2 : \tau_2 \dashv \ell_\mathrm{O}$, so LET is sufficient to prove $\Sigma_{\sigma'};\Gamma;pc;\ell_\mathrm{I} \vdash \mathsf{let}\ x = s'_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \ell_\mathrm{O}$, finishing the case.

Case VARIANCE**:** By induction on the typing rules.

Cases ATPC**,** WITHLOCK**, and** RETURN**:** Each of these three cases has two sub-cases: where the sub-statement is a value and where it is not. If the sub-statement is a value, the step must be E-ATPC, E-UNLOCK, or E-RETURN, respectively. In each case, VAL allows values to type-check with any *pc* and lock labels, proving the case. If the sub-statement is not a value, the only step possible is E-EVAL. Here the proof follows by induction on the typing rules in the same manner as the LET case above. □

Several semantic steps (E-REF, E-ASSIGN, and E-CALL) include information-security checks to guarantee that the code performing the operation is sufficiently trusted. The type system guarantees that these labels remain at least as trusted as the *pc* label of code executing. We formally define this property as a relation between a label stack and a statement, denoted by $\mathcal{M} \leftrightsquigarrow s$, and then prove that the semantics maintains this relation. The relation is formally defined on evaluation contexts and extended to statements $s = E[e]$ if $\mathcal{M} \leftrightsquigarrow E$.

$$\frac{}{\ell_m \leftrightsquigarrow [\cdot]} \qquad \frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow \texttt{let } x = E \texttt{ in } e} \qquad \frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow E \texttt{ with-lock } \ell} \qquad \frac{\mathcal{M} \leftrightsquigarrow E}{\mathcal{M} \leftrightsquigarrow \texttt{ignore-locks-in } E}$$

$$\frac{\mathcal{M} \leftrightsquigarrow E}{\ell, \mathcal{M} \leftrightsquigarrow \texttt{return}_\tau\ E} \qquad \frac{\ell, \mathcal{M} \leftrightsquigarrow E \qquad \ell \Rightarrow pc}{\ell, \mathcal{M} \leftrightsquigarrow E \texttt{ at-pc } pc}$$

Proposition 4.1. *For any statements s and s′ and configurations* $C = (CT, \sigma, (\ell_m, \mathcal{M}), L)$ *and* $C' = (CT, \sigma', \mathcal{M}', L')$*, if* $\vdash CT$ ok *and* $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ *and* $\Sigma_\sigma; \Gamma; \ell_m; \ell_I \vdash s : \tau \dashv \ell_O$ *and* $\langle s \mid C \rangle \longrightarrow \langle s' \mid C' \rangle$*, then* $\mathcal{M}' \leftrightsquigarrow s'$.

The proof of Proposition 4.1 relies on two lemmas.

Lemma 4.6. *For any label list* $\mathcal{M}$ *and evaluation contexts* E_1 *and* E_2*,* $\mathcal{M} \leftrightsquigarrow E_1[E_2]$ *if and* *only if there exist $\mathcal{M}_1$, $\mathcal{M}_2$, and ℓ_m such that (1) $\mathcal{M}_1, \ell_m, \mathcal{M}_2 = \mathcal{M}$, (2) $\mathcal{M}_1, \ell_m \leftrightsquigarrow E_1$, and (3) $\ell_m, \mathcal{M}_2 \leftrightsquigarrow E_2$.*

Proof. This is a proof by induction on E_1.

Case $E_1 = [\cdot]$:

($\Rightarrow$) Let $\mathcal{M}_1$ be empty and note that $\mathcal{M}$ cannot be empty, so $\mathcal{M} = \ell_m, \mathcal{M}_2$.

($\Leftarrow$) By inversion on the rules, $\mathcal{M}_1$ must be empty, so $\mathcal{M} = \ell_m, \mathcal{M}_2$, proving the result.

Case $E_1 = (\text{let } x = E_1' \text{ in } e)$, E_1' with-lock ℓ, or ignore-locks-in E_1':

($\Rightarrow$) By induction, there exist $\mathcal{M}_1$, $\mathcal{M}_2$, and ℓ_m such that $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2$, $\mathcal{M}_1, \ell_m \leftrightsquigarrow E_1'$, and $\ell_m, \mathcal{M}_2 \leftrightsquigarrow E_2$. Therefore, by the appropriate rule, $\mathcal{M}_1, \ell_m \leftrightsquigarrow E_1$.

($\Leftarrow$) By induction, $\mathcal{M}_1, \ell_m, \mathcal{M}_2 \leftrightsquigarrow E_1'[E_2]$, so the appropriate rule proves $\mathcal{M}_1, \ell_m, \mathcal{M}_2 \leftrightsquigarrow E_1[E_2]$.

Case $E_1 = \text{return}_\tau\ E_1'$:

($\Rightarrow$) Inversion on the correspondence proves $\mathcal{M} = \ell, \mathcal{M}'$ and $\mathcal{M}' \leftrightsquigarrow E_1'[E_2]$. By induction, there is some $\mathcal{M}_1', \ell_m, \mathcal{M}_2 = \mathcal{M}'$ such that $\mathcal{M}_1', \ell_m \leftrightsquigarrow E_1'$ and $\ell_m, \mathcal{M}_2 \leftrightsquigarrow E_2$. Letting $\mathcal{M}_1 = \ell, \mathcal{M}_1'$ completes the case.

($\Leftarrow$) By inversion on the correspondence rules, if $\mathcal{M}_1, \ell_m \leftrightsquigarrow E_1$, then $\mathcal{M}_1 = \ell, \mathcal{M}_1'$ for some ℓ and $\mathcal{M}_1'$ and $\mathcal{M}_1', \ell_m \leftrightsquigarrow E_1'$. By induction, $\mathcal{M}_1', \ell_m, \mathcal{M}_2 \leftrightsquigarrow E_1'[E_2]$, so therefore

$$\mathcal{M}_1, \ell_m, \mathcal{M}_2 = \ell, \mathcal{M}_1', \ell_m, \mathcal{M}_2 \leftrightsquigarrow \text{return}_\tau\ E_1'[E_2] = E_1[E_2].$$

Case $E_1 = E_1'$ at-pc pc:

($\Rightarrow$) By inversion on the rules, $\mathcal{M} \leftrightsquigarrow E_1'[E_2]$, so by induction $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2$ with the desired properties. Moreover, $\mathcal{M} = \ell, \mathcal{M}'$ and $\ell \Rightarrow pc$. Because

$\mathcal{M}_1, \ell_m$ is a non-empty prefix of $\mathcal{M}$, it must be the case that $\mathcal{M}_1, \ell_m = \ell, \mathcal{M}'_1$, so therefore $\mathcal{M}_1, \ell_m \leftrightsquigarrow E'_1$ at-pc $pc = E_1$, as desired.

($\Leftarrow$) By inversion on the correspondence rules, $\mathcal{M}_1, \ell_m \leftrightsquigarrow E'_1$, so by induction, $\mathcal{M} = \mathcal{M}_1, \ell_m, \mathcal{M}_2 \leftrightsquigarrow E'_1[E_2]$. Moreover, $\mathcal{M}_1, \ell_m = \ell, \mathcal{M}'_1$ and $\ell \Rightarrow pc$. Therefore $\mathcal{M} = \ell, \mathcal{M}'_1, \mathcal{M}_2$, satisfying the requirements to prove $\mathcal{M} \leftrightsquigarrow E'_1[E_2]$ at-pc $pc = E_1[E_2]$. □

Lemma 4.7. *For any statements s and s', configurations $C = (CT, \sigma, \mathcal{M}, L)$ and $C' = (CT, \sigma', \mathcal{M}', L')$, and label lists $\mathcal{M}_1$ and $\mathcal{M}_2$, if $\mathcal{M} = \mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}_2$ is not empty, then $\langle s \mid C \rangle \longrightarrow \langle s' \mid C' \rangle$ if and only if $\langle s \mid C[\mathcal{M}_2/\mathcal{M}] \rangle \longrightarrow \langle s' \mid C'[\mathcal{M}'_2/\mathcal{M}] \rangle$ for some $\mathcal{M}'_2$ where $\mathcal{M}' = \mathcal{M}_1, \mathcal{M}'_2$.*

Proof. By simple induction on the operational semantics. □

Proof of Proposition 4.1. This is a proof by induction on the operational semantics.

Case E-EVAL**:** In this case $s = E[\tilde{s}]$, and by definition, $\tilde{s} = \tilde{E}[e]$. By Lemma 4.6, there exist $\mathcal{M}_1$, $\mathcal{M}_2$, and ℓ such that $\ell_m, \mathcal{M} = \mathcal{M}_1, \ell, \mathcal{M}_2$ where $\mathcal{M}_1, \ell \leftrightsquigarrow E$ and $\ell, \mathcal{M}_2 \leftrightsquigarrow \tilde{E}$. Therefore E-EVAL gives $\langle \tilde{s} \mid C \rangle \longrightarrow \langle \tilde{s}' \mid C' \rangle$, and because $\ell, \mathcal{M}_2$ is non-empty, Lemma 4.7 proves $\langle \tilde{s} \mid C[(\ell, \mathcal{M}_2)/\mathcal{M}] \rangle \longrightarrow \langle \tilde{s}' \mid C'[\mathcal{M}'_2/\mathcal{M}] \rangle$, and moreover $\mathcal{M}' = \mathcal{M}_1, \mathcal{M}'_2$. Induction on this step ensures that $\mathcal{M}'_2 \leftrightsquigarrow \tilde{s}'$, so therefore $\mathcal{M}'_2$ must be non-empty. As a single step can only add or remove one element from $\mathcal{M}$, that means $\mathcal{M}'_2 = \ell, \mathcal{M}''_2$, so by Lemma 4.6, $\mathcal{M}' = \mathcal{M}_1, \ell, \mathcal{M}''_2 \leftrightsquigarrow E[\tilde{s}'] = s'$.

Case E-IFT **and** E-IFF**:** Here $s =$ if$\{pc\}$ v then e_1 else e_2. By inversion on the correspondence rules, $\mathcal{M} = \cdot$, and by inversion on the typing rules $\ell_m \Rightarrow pc$. Therefore $\ell_m \leftrightsquigarrow [\cdot]$ at-pc pc, so by definition $\mathcal{M}' = \ell_m \leftrightsquigarrow (e_i$ at-pc $pc) = s'$.

Case E-ATPC**:** Here $s = v$ at-pc pc and $s' = v$. By inversion on the correspondence rules, $\mathcal{M} = \cdot$ and $\mathcal{M}' = \ell_m$. Because $\ell_m \leftrightsquigarrow v$ for any v, this completes the case.

Cases E-CALL **and** E-CALLATK**:** In both of these cases, $s = \text{new } C(\overline{v}).m(\overline{w})$ and $mbody(C, m) = (\ell'_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)$. By inversion on the correspondence rules, $\mathcal{M} = \cdot$ and $\mathcal{M}' = \ell_m, \ell'_m$. By METHOD-OK, $\ell'_m \Rightarrow pc_2$. Therefore, letting $e' = e[\overline{x} \mapsto \overline{w}, \text{this} \mapsto \text{new } C(\overline{v})]$,

$$\dfrac{\dfrac{\dfrac{}{\ell'_m \leftrightsquigarrow e'} \qquad \ell'_m \Rightarrow pc_2}{\ell'_m \leftrightsquigarrow e' \text{ at-pc } pc_2}}{\ell_m, \ell'_m \leftrightsquigarrow \text{return}_\tau\ (e' \text{ at-pc } pc_2)}\ .$$

Case E-RETURN**:** Here $s = \text{return}_\tau\ v$, so inversion on the correspondence rules proves $\mathcal{M} = \ell$. Therefore $\mathcal{M}' = \ell_m \leftrightsquigarrow v = s'$ proves the case.

No other operational semantic rules modify $\mathcal{M}$ or add or remove return or at-pc terms. Therefore the same proofs apply before and after the step. □

The progress theorem is not without caveats. SeRIF's type system intentionally leaves checking of explicit casts, null dereferences, and dynamic reentrancy locks to run time. As a result, the progress theorem states that these three are the *only* ways a well-typed program can get stuck.

Theorem 4.7 (Progress)**.** *For a statement s and configuration $C = (CT, \sigma, (\ell_m, \mathcal{M}), L)$, if*

- $\Sigma_\sigma; \cdot; pc; \ell_I \vdash s : \tau \dashv \ell_O$,
- $\ell_m \Rightarrow pc$, *and*
- $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$,

then one of the following holds:

1. *s is a closed value,*
2. *$\langle s \mid C \rangle \longrightarrow \langle s' \mid C' \rangle$ for some s' and C',*
3. *$s = E[(C)(\text{new } D(\overline{v}))]$ where $D \not<: C$,*
4. *$s = E[!\text{null}]$ or $s = E[\text{null} := v]$, or*

5. *$s = E[\textsf{new}\ C(\overline{v}).m(\overline{w})]$ for a C and m such that $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \tau$ and there is some $\ell'_m \in L$ such that $pc_1 \not\Rightarrow pc_2 \vee \ell'_m$.*

Proof. This is a proof by induction on the derivation that $\Sigma_\sigma; \cdot; pc; \ell_I \vdash s : \tau \dashv \ell_O$.

Case VAL**:** Because $\Gamma = \cdot$, s is a closed value.

Case ENDORSE**:** Here $s = \textsf{endorse}\ v\ \textsf{from}\ \ell\ \textsf{to}\ \ell'$. Since $\Gamma = \cdot$, v is a closed value, so E-ENDORSE applies.

Case CAST**:** Here $s = (C)v$. Inversion on the value typing rules coupled with the fact that $\Gamma = \cdot$ proves that $v = \textsf{new}\ D(\overline{v})$. If $D <: C$, then E-CAST applies with $C' = C$. Otherwise this is a bad cast.

Case FIELD**:** Here $s = v.f_i$. Again, inversion on the value typing rules with $\Gamma = \cdot$ proves $v = \textsf{new}\ C(\overline{v})$. Moreover FIELD requires reference to a valid fields, so E-FIELD steps s with $C' = C$.

Case CALL**:** Here $s = v.m(\overline{v})$. Any step must be either E-CALL or E-CALLATK. Because $\Gamma = \cdot$, inversion on the premise that $\Sigma_\sigma; \Gamma \vdash v : C^\ell$ proves $v = \textsf{new}\ C(\overline{w})$. The premise $\Sigma; \Gamma \vdash \overline{v} : \overline{\tau_a}$ also directly proves the corresponding premise of E-CALL/E-CALLATK. Inversion on the proof that $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ proves that $\mathcal{M}$ is empty, so therefore the premise of E-CALL/E-CALLATK requiring the caller's integrity to act for pc_1 is satisfied by $\ell_m \Rightarrow pc \Rightarrow pc_1$. At this point, E-CALLATK applies if $\ell_{\mathcal{A}} \Rightarrow pc_2$ and E-CALL applies if $\bigwedge_{\ell \in L}(pc_1 \Rightarrow pc_2 \vee \ell)$. Therefore, if the statement is stuck, neither is satisfied, and the second is precisely the condition of a dynamic reentrancy lock blocking a call.

Case IF**:** Here $s = \textsf{if}\{pc'\}\ v\ \textsf{then}\ e_1\ \textsf{else}\ e_2$. Inversion on the value typing rules using $\Gamma = \cdot$ means $v = \textsf{true}$ or $v = \textsf{false}$. Therefore E-IFT or E-IFF apply.

Case REF**:** Here $s = \textsf{ref}\ v\ \tau$. This step will be with E-REF. Since $\Gamma = \cdot$, the requirement that $\Sigma_\sigma \vdash v : \tau$ comes directly from REF. Moreover, inversion on the rules proving $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ shows that $\mathcal{M} = \cdot$, so the protection requirement of E-Ref is $\ell_m \triangleleft \tau$ and $\ell_m \Rightarrow pc \triangleleft \tau$, meaning the step applies with some fresh $\iota \notin \mathrm{dom}(\sigma)$.

Case Deref**:** Here $s = !v$. Since $\Gamma = \cdot$, inversion on the Deref premise that $\Sigma \vdash v : (\mathsf{ref}\ \tau')^{\ell}$ means $v = \iota$ with $\Sigma_\sigma(\iota) = \tau'$ or $v = \mathsf{null}$. In the first case, by definition this means $\sigma(\iota) = (v', \tau')$ for some v', so E-Deref applies. In this second case, this is a null dereference.

Case Assign**:** Here $s = (v_1 := v_2)$. Again, $\Gamma = \cdot$ and inversion on the typing rules using the premise $\Sigma; \Gamma \vdash v_1 : (\mathsf{ref}\ \tau)^{\ell}$ proves that $v_1 = \iota$ or $v_1 = \mathsf{null}$. If $v_1 = \mathsf{null}$, then this is a null dereference. If $v_1 = \iota$, then the step must be E-Assign. The requirement that $\Sigma_\sigma(\iota) = \tau$ and $\Sigma \vdash v_2 : \tau$ stem from inversion on the typing derivation of v_1 and the second premise of Assign. Finally, inversion on the rules proving $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ shows that $\mathcal{M} = \cdot$, and Assign requires $pc \vee \ell \triangleleft \tau$, so the transitivity of $\Rightarrow$ proves $\ell_m \triangleleft \tau$, as needed.

Case Lock**:** E-Lock always applies.

Case Let**:** Here $s = \mathsf{let}\ x = \tilde{s}\ \mathsf{in}\ e$. The first hypothesis of Let proves $\Sigma_\sigma; \cdot; pc; \ell_{\mathrm{I}} \vdash \tilde{s} : \tau_1 \dashv \ell'_{\mathrm{O}}$, and $(\ell_m, \mathcal{M}) \leftrightsquigarrow \tilde{s}$. Therefore, our inductive hypothesis applies to $\tilde{s}$. If $\tilde{s}$ is a closed value, then E-Let applies to s, stepping to $s' = e[x \mapsto \tilde{s}]$ letting $C' = C$. If $\langle \tilde{s} \mid C \rangle \longrightarrow \langle \tilde{s}' \mid C' \rangle$, then by E-Eval, $\langle s \mid C \rangle \longrightarrow \langle \mathsf{let}\ x = \tilde{s}'\ \mathsf{in}\ e \mid C' \rangle$. For the other three cases where $\tilde{s} = E[e']$ where e' is a failure condition, we note that $\mathsf{let}\ x = E\ \mathsf{in}\ e$ is an evaluation context, so s falls into the same failure case.

Case Variance**:** Because $\ell_m \Rightarrow pc \Rightarrow pc'$, this case follows directly by induction.

Case AtPc**:** Here $s = \tilde{s}\ \mathsf{at\text{-}pc}\ pc'$. Inversion on the proof that $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ proves $\ell_m \Rightarrow pc'$. The hypothesis of AtPc is $\Sigma_\sigma; \cdot; pc'; \ell_{\mathrm{I}} \vdash \tilde{s} : \tau \dashv \ell_{\mathrm{O}}$, so the inductive hypothesis applies to $\tilde{s}$.

If $\tilde{s}$ is a closed value, then E-AtPc applies letting $C' = C$. If $\tilde{s}$ steps to $\tilde{s}'$, then E-Eval proves $\langle s \mid C \rangle \longrightarrow \langle \tilde{s}' \text{ at-pc } pc' \mid C' \rangle$. For the other three cases, as with Let, $\tilde{s} = E[e]$ where e is a failure condition, so $E \text{ at-pc } pc'$ is an evaluation context proving that s falls into the same failure case as $\tilde{s}$.

Cases WithLock **and** IgnoreLocks**:** The logic of these cases is the same as the logic of the AtPc case, but using pc instead of pc'.

Case Return**:** Here $s = \text{return}_\tau\ \tilde{s}$. Inversion on the proof that $(\ell_m, \mathcal{M}) \leftrightsquigarrow s$ shows that $\mathcal{M}$ is not empty and $\mathcal{M} \leftrightsquigarrow \tilde{s}$. Additionally, a premise of Return is $\Sigma_\sigma; \cdot; pc; \ell'_\text{I} \vdash \tilde{s} : \tau \dashv \ell'_\text{O}$. Therefore, the inductive hypothesis applies using Lemma 4.7 to replace $(\ell_m, \mathcal{M})$ with $\mathcal{M}$ in C.

If $\tilde{s}$ is a closed value, the well-typed premise of Return proves $\Sigma_\sigma \vdash v : \tau$, and since $(\ell_m, \mathcal{M})$ is non-empty, E-Return applies. If $\tilde{s}$ steps to $\tilde{s}'$, then E-Eval allows s to step as well. Again, for the three failure cases where $\tilde{s} = E[e]$, simply replacing E with $\text{return}_\tau\ E$ creates the expected form. □

Note that, for any invocation $I = (\ell, \iota, m(\overline{v}))$, $\ell \leftrightsquigarrow\ !\iota.m(\overline{v})$. Therefore, if the invocation and class table are well-typed in Σ_σ for a well-typed heap σ, Theorems 4.6 and 4.7 combine with Proposition 4.1 to prove that the invocation steps to a closed value with a well-typed heap, diverges (can take infinitely many steps), or gets stuck on one of the three run-time error checks.

4.11 Proof of Noninterference

We now prove Theorem 4.1, presented in Section 4.5, using an erasure-based construction. Specifically, we will erase low-integrity values in the heap and then execute the same program using a modified semantics that continues to omit low-integrity values from the state and uses a special value, $\bullet$, when one would be read. We prove that, if the original execution terminated and the code is endorsement-free, this modified execution must terminate and, critically, the high-integrity components of the state must match. The theorem then follows by noting that if $\sigma_1 \approx_{\ell_t} \sigma_2$, then both executions must produce heaps that's high-integrity components are the same as the modified execution on a partially-erased heap.

Formally, we introduce a new value to denote erased data.

$$v \quad ::= \quad \cdots \mid \bullet$$

The typing and semantic rules that handle $\bullet$ are parameterized on a label ℓ_t defining high-integrity values. For notational ease, we omit that label in our syntax. However, as our theorems are all parameterized over ℓ_t, they remain true for any possible choice of ℓ_t.

The type system allows $\bullet$ to be any type, as long as that type is low-integrity. To simplify notation, we define $\mathrm{label}(t^{\ell}) = \ell$.

$$[\textsc{Bullet}]\ \frac{\mathrm{label}(\tau) \not\Rightarrow \ell_t}{\Sigma; \Gamma \vdash \bullet : \tau}$$

We introduce a new operational semantics in Figure 4.11 to deal with these terms. To separate executions with and without bullets, we define a new step function denoted $\multimap\!\!\!\rightarrow$ when working with erased terms. We also define a context B defining syntactic forms that normally require a decision based on the value. The B-BULLETCTX rule simply erases the entire expression when the given value is $\bullet$.

$$B \quad ::= \quad \mathsf{if}\{pc\}\ [\cdot]\ \mathsf{then}\ e\ \mathsf{else}\ e \mid\ ![\cdot] \mid (C)[\cdot] \mid [\cdot].f \mid [\cdot].m(\overline{v})$$

These semantics inherit from our original operation semantics whenever the step does not modify the heap. When modifying the heap, however, $\multimap\!\!\!\rightarrow$ omits any

$$[\text{B-PureStep}]\ \frac{\langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow \langle s' \mid (CT, \sigma, \mathcal{M}', L')\rangle}{\langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle s' \mid (CT, \sigma, \mathcal{M}', L')\rangle}$$

$$[\text{B-Eval}]\ \frac{\langle s \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle s' \mid C'\rangle}{\langle E[s] \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle E[s'] \mid C'\rangle} \qquad [\text{B-BulletCtx}]\ \frac{}{\langle B[\bullet] \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle \bullet \mid C\rangle}$$

$$[\text{B-TRef}]\ \frac{\begin{array}{c}\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \\ \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau \qquad \mathrm{label}(\tau) \Rightarrow \ell_t\end{array}}{\langle \mathsf{ref}\ v\ \tau \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle \iota \mid C[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

$$[\text{B-URef}]\ \frac{\begin{array}{c}\iota \notin \mathrm{dom}(\sigma) \qquad \Sigma_\sigma \vdash v : \tau \\ \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau \qquad \mathrm{label}(\tau) \not\Rightarrow \ell_t\end{array}}{\langle \mathsf{ref}\ v\ \tau \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle \iota \mid C[\sigma[\iota \mapsto (\bullet, \tau)]/\sigma]\rangle} \qquad [\text{B-BAssign}]\ \frac{}{\langle \bullet := v \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle () \mid C\rangle}$$

$$[\text{B-TAssign}]\ \frac{\begin{array}{c}\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \\ \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau \qquad \mathrm{label}(\tau) \Rightarrow \ell_t\end{array}}{\langle \iota := v \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle () \mid C[\sigma[\iota \mapsto (v, \tau)]/\sigma]\rangle}$$

$$[\text{B-UAssign}]\ \frac{\begin{array}{c}\Sigma_\sigma(\iota) = \tau \qquad \Sigma_\sigma \vdash v : \tau \\ \mathcal{M} = \mathcal{M}', \ell_m \qquad \ell_m \lhd \tau \qquad \mathrm{label}(\tau) \not\Rightarrow \ell_t\end{array}}{\langle \iota := v \mid C\rangle \mathrel{-\!\!\bullet\!\!\!\rightarrow} \langle () \mid C[\sigma[\iota \mapsto (\bullet, \tau)]/\sigma]\rangle}$$

Figure 4.11: Small-step operational semantics for SeRIF with bullets.

values that are in low-integrity memory locations, while treating high-integrity memory locations normally. When reading from the heap, it produces $\bullet$ whenever it tries to read from a location that has a type but not a value. In our construction for our proof, these will be precisely the low-integrity locations.

We now claim that, if CT is endorsement-free at ℓ_t, then any invocation with input state σ_1 will, under normal semantics, produce a state σ_2 that is ℓ_t-equivalent to executing the same invocation under bullet semantics with input state $\sigma_1|_{\ell_t}$.

Lemma 4.8 (Label Stack Maintenance). *For any expression e, if*

$$\langle e \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L')\rangle,$$

then $\mathcal{M}' = \mathcal{M}$ and $L' = L$.

Proof. This will be a proof by induction on the number of steps from e to v and on the operational semantics. In the base case, there are zero steps, so the result trivially holds.

We now assume $\langle e \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L')\rangle$ takes $n \geq 1$ steps and the result holds for all executions of $k < n$ steps. We consider the following cases.

Case E-EVAL**:** If $E = [\cdot]$, we can replace this step with another, so without loss of generality, we assume $E \neq [\cdot]$. Since $e = E[\tilde{e}]$ is an expression, $\tilde{e}$ is also an expression and $E = \mathsf{let}\ x = x\ \mathsf{in}\ E'e''$. By inversion on the operational semantics, E-EVAL is the only rule that can apply until $\tilde{e}$ reaches some value $\tilde{v}$. Moreover, $E[\tilde{v}]$ is not a value, so $\langle \tilde{e} \mid C\rangle \longrightarrow^* \langle \tilde{v} \mid \tilde{C}\rangle$ in fewer steps. By induction, we therefore have that $\tilde{\mathcal{M}} = \mathcal{M}$ and $\tilde{L} = L$. Moreover, since $E[\tilde{e}]$ was surface syntax, $E[\tilde{v}]$ must be as well. Therefore, another application of our inductive hypothesis proves

$$\langle E[\tilde{e}] \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow^* \langle E[\tilde{v}] \mid (CT, \tilde{\sigma}, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}, L)\rangle$$

Cases E-IFT **and** E-IFF**:** Because $e = \mathsf{if}\{pc\}\ v'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ was surface-syntax, e_i must also be surface syntax. Inspection on the semantic rules says that any expression of the form $\tilde{e}\ \mathsf{at\text{-}pc}\ pc$ can only step using E-EVAL if $\tilde{e}$ steps or E-ATPC if $\tilde{e}$ is a value. Therefore, we know that $\langle \tilde{e} \mid C\rangle \longrightarrow^* \langle v \mid C'\rangle$, and then $v\ \mathsf{at\text{-}pc}\ pc$ steps once using E-ATPC. By induction on the number of steps, we therefore have that $\mathcal{M}' = \mathcal{M}$ and $L' = L$.

Case E-LOCK**:** This case is similar to the previous case. Here $e = \mathsf{lock}\ \tilde{e}\ \mathsf{in}\ \ell$ and $\tilde{e}$ is surface-syntax. We also know that $\langle e \mid C\rangle \longrightarrow \langle \tilde{e}\ \mathsf{with\text{-}lock}\ \ell \mid C[L, \ell/L]\rangle$. By the same argument as above, $\tilde{e}$ must step to a value in fewer steps, so by induction

$$\langle \tilde{e} \mid (CT, \sigma, \mathcal{M}, (L, \ell))\rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}, (L, \ell))\rangle$$

A single application of E-UNLOCK then gives us the desired result.

Cases E-CALL **and** E-CALLATK**:** These cases are identical to the previous one, but modifying $\mathcal{M}$ instead of L and using E-RETURN instead of E-UNLOCK.

Cases E-UNLOCK **and** E-RETURN**:** These are impossible because e is surface-syntax.

In all other cases, stepping e once continues to be surface syntax and leaves $\mathcal{M}$ and L unmodified. We can therefore remove a single step and apply our inductive hypothesis. □

Lemma 4.9 (Step Confinement)**.** *For a state σ_1 where $\Sigma \subseteq \Sigma_{\sigma_1}$ and a statement s_1, if*

1. $\Sigma \vdash CT$ ok *is endorsement-free at ℓ_t,*
2. $\vdash \sigma_1$ wt,
3. $\Sigma; \Gamma; pc; \ell_I \vdash s_1 : \tau \dashv \ell_O$,
4. $pc \not\Rightarrow \ell_t$,
5. *for all sub-statements s* at-pc *pc' of s_1, $pc' \not\Rightarrow \ell_t$, and*
6. $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}_1, L_1)\rangle \longrightarrow \langle s_2 \mid (CT, \sigma_2, \mathcal{M}_2, L_2)\rangle$,

then

- $\sigma_1 \approx_{\ell_t} \sigma_2$ *and*
- *for all sub-statements s* at-pc *pc' of s_2, $pc' \not\Rightarrow \ell_t$.*

Proof. This will be a proof by induction on the semantic rule used to take a step. The following are the nontrivial cases.

Case E-EVAL**:** In this case $s_1 = E[\tilde{s}_1]$. We claim by induction on E that, for some pc', ℓ'_I, τ', and ℓ'_O where $pc' \not\Rightarrow \ell_t$, $\Sigma; \Gamma; pc'; \ell'_I \vdash \tilde{s}_1 : \tau' \dashv \ell'_O$. If $E = [\cdot]$, this follows directly from our assumptions. If $E = \text{let } x = x \text{ in } E' s'$, $\text{return}_\tau\ E'$, or E' with-lock ℓ, we note that $\Sigma; \Gamma; pc; \ell'_I \vdash E'[\tilde{s}_1] : \tau' \dashv \ell'_O$ for some ℓ'_I, τ', and ℓ'_O, so by induction on E, we have the desired result. If $E = E'$ at-pc pc'', we note that $\Sigma; \Gamma; pc''; \ell'_I \vdash E'[\tilde{s}_1] : \tau \dashv \ell_O$ and, by assumption, $pc'' \not\Rightarrow \ell_t$. Thus induction on E again gets us the desired typing judgment.

E-EVAL tells us $\langle \tilde{s}_1 \mid (CT, \sigma_1, \mathcal{M}_1, L_1) \rangle \longrightarrow \langle \tilde{s}_2 \mid (CT, \sigma_2, \mathcal{M}_2, L_2) \rangle$ and $s_2 = E[\tilde{s}_2]$. Since $\tilde{s_1}$ is a sub-statement of s_1, it must satisfy hypothesis 5, and the typing judgment above gives us hypotheses 3 and 4. Induction on the operational semantics therefore gives us $\sigma_1 \approx_{\ell_t} \sigma_2$ and, for all sub-statements e $\mathsf{at\text{-}pc}$ pc' of $\tilde{s}_2$, $pc' \not\Rightarrow \ell_t$. By hypothesis 5, the same must be true of E, so therefore $E[\tilde{s}_2] = s_2$ satisfies the required condition.

Cases E-IFT **and** E-IFF**:** Here $s_1 = \mathsf{if}\{pc'\}\ v\ \mathsf{then}\ s_1'\ \mathsf{else}\ s_2'$. We know that $\sigma_1 = \sigma_2$, so that condition is trivially true. Both s_1' and s_2' are surface-syntax, so they contain no sub-statement of the form e $\mathsf{at\text{-}pc}$ pc'', meaning the only such sub-statement in $s_2 = s_i'$ $\mathsf{at\text{-}pc}$ pc' is the outer one. By inversion on the typing rules, we know that $pc \Rightarrow pc'$, so by transitivity, $pc' \not\Rightarrow \ell_t$.

Case E-REF**:** Here $s_1 = \mathsf{ref}\ v\ \tau'$. By inversion on the typing rules, we know that $pc \triangleleft \tau'$, and by assumption, $pc \not\Rightarrow \ell_t$. Therefore, since $\iota \notin \mathrm{dom}(\sigma_1)$ and $\sigma_2 = \sigma_1[\iota \mapsto (v, \tau')]$, we know that $\sigma_1|_{\ell_t} = \sigma_2|_{\ell_t}$, which is exactly the definition of $\sigma_1 \approx_{\ell_t} \sigma_2$. There are no sub-statements of the form e $\mathsf{at\text{-}pc}$ pc', so that result is trivially true.

Case E-ASSIGN**:** Here $s_1 = \iota := v$. By inversion on the typing rules, we know $\Sigma(\iota) = \tau'$ and $pc \triangleleft \tau'$. By assumption, $pc \not\Rightarrow \ell_t$, so therefore $\iota \notin \mathrm{dom}((|_{\ell_t}\sigma_1))$. Given this and the fact that $\Sigma_{\sigma_2} = \Sigma_{\sigma_1}$, again $\sigma_1 \approx_{\ell_t} \sigma_2$, as desired. As in the previous case, there are no sub-statements of the form e $\mathsf{at\text{-}pc}$ pc'.

Cases E-CALL **and** E-CALLATK**:** In both of these cases, $s_1 = \mathsf{new}\ C(\overline{v}).m(\overline{w})$ with $mbody(C, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)$. Inversion on the typing rules proves that $pc \Rightarrow pc_1$, and by assumption, $pc \not\Rightarrow \ell_t$. Therefore, by transitivity, $pc_1 \not\Rightarrow \ell_t$, so, by the definition of CT being endorsement-free at ℓ_t, it must be the case that $pc_2 \not\Rightarrow \ell_t$. Moreover, $e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})]$ is surface-syntax, so the only sub-statement of the form s' $\mathsf{at\text{-}pc}$ pc' on s_2 is the outer one where $pc' =$

pc_2, and we just proved $pc_2 \not\Rightarrow \ell_t$. Finally, the step leaves the heap and heap type unmodified, finishing the case.

All other cases leave the heap and heap type unmodified and do not add sub-statement of the form e at-pc pc', making the result trivial in those cases. $\square$

Corollary 4.1 (Confinement)**.** *Given a class table CT and an expression (not statement) e, if*

- *CT and e are both endorsement-free at ℓ_t,*
- $\Sigma_\sigma \vdash CT$ ok,
- $\Sigma_\sigma; \Gamma; pc; \ell_\mathrm{I} \vdash e : \tau \dashv \ell_\mathrm{O}$ *for some $pc \not\Rightarrow \ell_t$, and*
- $\langle e \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid (CT, \sigma', \mathcal{M}', L')\rangle$,

then $\sigma \approx_{\ell_t} \sigma'$, $\mathcal{M} = \mathcal{M}'$, and $L = L'$.

Proof. We apply Lemma 4.8 and inductively apply Lemma 4.9 using the fact that expressions cannot contain any subexpressions of the form e' at-pc pc'. $\square$

We aim to prove something about execution in our regular semantics through execution in our semantics with bullets, so we need a way to relate terms with and without bullets. We do this using a syntactic relation denoted $e_1 \geq_\bullet e_2$ to indicate that e_2 is just e_1 but possibly with some information erased. On values, the relation is defined as follows.

$$\frac{v \neq \bullet}{v \geq_\bullet v} \qquad \frac{v \notin \{x, \bullet\}}{v \geq_\bullet \bullet} \qquad \frac{\overline{v} \geq_\bullet \overline{w}}{\text{new } C(\overline{v}) \geq_\bullet \text{new } C(\overline{w})}$$

We extend this relation to typing proofs. We first relate typing proofs of closed values (so proofs that do not use VAR) to value typing proofs using BULLET. Note that we do not mandate that the heap types be the same at every location so long as they are the same at the locations used in the typing proof. That is

$$[\textsc{Loc}]\ \frac{\Sigma_1(\iota) = \tau}{\Sigma_1; \Gamma \vdash \iota : (\mathsf{ref}\ \tau)^{\ell}} \geq_{\bullet} \frac{\Sigma_2(\iota) = \tau}{\Sigma_2; \Gamma \vdash \iota : (\mathsf{ref}\ \tau)^{\ell}}\ [\textsc{Loc}]$$

whenever $\Sigma_1(\iota) = \Sigma_2(\iota)$, though if Σ_1 and Σ_2 may differ on other locations. Notably, if $\Sigma_1(\iota) = \tau \neq \Sigma_2(\iota)$ (possibly because $\iota \notin \mathrm{dom}(\Sigma_2)$) and $\ell \not\Rightarrow \ell_t$, then,

$$[\textsc{Loc}]\ \frac{\Sigma_1(\iota) = \tau}{\Sigma_1; \Gamma \vdash \iota : (\mathsf{ref}\ \tau)^{\ell}} \geq_{\bullet} \frac{\ell \not\Rightarrow \ell_t}{\Sigma_2; \Gamma \vdash \bullet : (\mathsf{ref}\ \tau)^{\ell}}\ [\textsc{Bullet}].$$

We finally extend the relation to typing proofs of expressions and statements by extending it structurally. That is, if the typing proofs of each sub-statement is related, then the typing proof of the whole statement is related. For example,

$$\frac{\dfrac{\pi_1}{\Sigma_1; \Gamma \vdash v_1 : \tau} \geq_{\bullet} \dfrac{\pi_2}{\Sigma_2; \Gamma \vdash v_2 : \tau}}{\dfrac{\dfrac{\pi_1}{\Sigma_1; \Gamma \vdash v_1 : \tau} \qquad pc \triangleleft \tau}{\Sigma_1; \Gamma; pc; \ell_{\mathrm{I}} \vdash \mathsf{ref}\ v\ \tau : (\mathsf{ref}\ \tau)^{\ell} \dashv \ell_{\mathrm{O}}} \geq_{\bullet} \dfrac{\dfrac{\pi_2}{\Sigma_2; \Gamma \vdash v_2 : \tau} \qquad pc \triangleleft \tau}{\Sigma_2; \Gamma; pc; \ell_{\mathrm{I}} \vdash \mathsf{ref}\ v\ \tau : (\mathsf{ref}\ \tau)^{\ell} \dashv \ell_{\mathrm{O}}}}$$

We usually denote this relation $\Sigma_1; \Gamma; pc; \ell_{\mathrm{I}} \vdash s_1 : \tau \dashv \ell_{\mathrm{O}} \geq_{\bullet} \Sigma_2; \Gamma; pc; \ell_{\mathrm{I}} \vdash s_2 : \tau \dashv \ell_{\mathrm{O}}$.

We now use this relation to relate executions in the regular semantics and the erasure semantics. For this we use a slightly modified erasure procedure on heaps, $\sigma|^{\bullet}_{\ell_t}$. Instead of simply removing all low-integrity mappings, it instead replaces the values with $\bullet$.

$$\sigma|^{\bullet}_{\ell_t}(\iota) \triangleq \begin{cases} (v, t^{\ell}) & \text{if } \sigma(\iota) = (v, t^{\ell}) \text{ and } \ell \Rightarrow \ell_t \\ (\bullet, t^{\ell}) & \text{if } \sigma(\iota) = (v, t^{\ell}) \text{ and } \ell \not\Rightarrow \ell_t \end{cases}$$

Lemma 4.10 (Bullet Semantics Completeness). *Let $C_i = (CT, \sigma_i, \mathcal{M}, L)$. If*

- $\Sigma_{\sigma_1}; \Gamma; pc; \ell_{\mathrm{I}} \vdash s_1 : \tau \dashv \ell_{\mathrm{O}} \geq_{\bullet} \Sigma_{\sigma_2}; \Gamma; pc; \ell_{\mathrm{I}} \vdash s_2 : \tau \dashv \ell_{\mathrm{O}}$,

- $\langle s_1 \mid C_1 \rangle \longrightarrow \langle s_1' \mid C_1' \rangle$,

then $\langle s_2 \mid C_2 \rangle \twoheadrightarrow\!\!\!\!\bullet\ \langle s_2' \mid C_2' \rangle$.

Proof. This is a proof by induction on the operational semantics of $s_1 \longrightarrow s_1'$.

Case E-EVAL**:** By induction on E, if $s_1 = E[\tilde{s}_1]$, then $s_2 = E'[\tilde{s}_2]$ where $\tilde{s}_1 \geq_\bullet \tilde{s}_2$. By induction on the operational semantics, $\langle \tilde{s}_2 \mid C_2 \rangle \twoheadrightarrow\!\!\!\!\bullet\ \langle \tilde{s}_2' \mid C_2' \rangle$, and B-EVAL applies to complete the case.

Cases E-IFT **and** E-IFF**:** Here the statement $s_1 = \mathsf{if}\{pc\}\ \tilde{v}\ \mathsf{then}\ \tilde{e}_1\ \mathsf{else}\ \tilde{e}_2$, and the statement $s_2 = \mathsf{if}\{pc\}\ v^\bullet\ \mathsf{then}\ e_1^\bullet\ \mathsf{else}\ e_2^\bullet$. We consider two sub-cases. First, if $v^\bullet = \bullet$, we see that $\langle s_2 \mid C_2 \rangle \twoheadrightarrow\!\!\!\!\bullet\ \langle \bullet \mid C_2 \rangle$ by B-BULLETCTX. If $v^\bullet \neq \bullet$, then $v^\bullet = \tilde{v}$, and therefore B-PURESTEP allows s_2 to step, as desired.

Cases E-CAST, E-FIELD, E-CALL, **and** E-CALLATK**:** These cases follow the same logic as the previous case, with their corresponding syntax.

Case E-REF**:** Here we have $s_1 = \mathsf{ref}\ v_1\ \tau$ so therefore $s_2 = \mathsf{ref}\ v_2\ \tau$ where $v_1 \geq_\bullet v_2$. Inversion on E-REF proves $\mathcal{M} = \mathcal{M}', \ell_m$ where $\ell_m \triangleleft \tau$. Since $\mathcal{M}$ is the same in C_1 and C_2, if $\mathrm{label}(\tau) \Rightarrow \ell_t$, then B-TREF applies, and if not, B-UREF applies.

Case E-DEREF**:** Here $s_1 = !\iota$ and $s_2 = !v$ where either $v = \bullet$ or $v = \iota$. If $v = \bullet$, then B-BULLETCTX applies. Otherwise, we know that $\Sigma_{\sigma_2}; \Gamma; pc, \ell_\mathrm{I} \vdash !\iota : \tau \dashv \ell_\mathrm{O}$. By inversion on the expression typing rules, we know that $\Sigma_{\sigma_2}; \Gamma \vdash \iota : (\mathsf{ref}\ \tau)^\ell$, and by inversion on the value typing rules, we therefore have $\Sigma_{\sigma_2}(\iota) = \tau$. In other words, $\iota \in \mathrm{dom}(\sigma_2)$, so B-PURESTEP applies with E-DEREF.

Case E-ASSIGN**:** Here $s_1 = \iota := v$ and $s_2 = v_1 := v_2$ where $v_1 = \bullet$ or $v_1 = \iota$. If $v_1 = \bullet$, then B-BASSIGN applies. Otherwise, because s_1 is well-typed with Σ_{σ_1}, inversion on the typing rules proves $\Sigma_{\sigma_1}(\iota) = \tau'$. Because s_1 steps with E-ASSIGN, inversion on E-ASSIGN proves $\mathcal{M} = \mathcal{M}', \ell_m$ where $\ell_m \triangleleft \tau'$. By inversion on the $\geq_\bullet$ relation, it must be the case that $\Sigma_{\sigma_2}(\iota) = \tau'$ and $\Sigma_{\sigma_2}; \Gamma \vdash v_2 : \tau'$. Since

$\mathcal{M}$ is the same in C_1 and C_2, this is sufficient to apply one of B-TASSIGN or B-UASSIGN, depending on label(τ').

For all other cases, the heap remains unmodified and no decisions are made based on a value that may be $\bullet$, so B-PURESTEP applies to s_2 using the same step that applied to s_1. □

Lemma 4.11 (Bullet Step Correspondence)**.** *For any class table CT, statements s_1 and s_2, heaps σ_1 and σ_2, and heap-type Σ, if*

- $\Sigma \vdash CT$ ok *is endorsement-free at ℓ_t,*
- *s_1 and s_2 are endorsement-free at ℓ_t,*
- $\vdash \sigma_i$ wt *for both $i = 1, 2$,*
- $\sigma_1 \approx_{\ell_t} \sigma_2$ *with* $\sigma_2 \subseteq \sigma_1|^{\bullet}_{\ell_t}$,
- $\Sigma \subseteq \Sigma_{\sigma_2}$,
- $\Sigma_{\sigma_1}; \Gamma; pc; \ell_I \vdash s_1 : \tau \dashv \ell_O \geq_{\bullet} \Sigma_{\sigma_2}; \Gamma; pc; \ell_I \vdash s_2 : \tau \dashv \ell_O$, *and*
- $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \longrightarrow^{+} \langle v \mid C\rangle$,

then there exists statements s_1' and s_2', heaps σ_1' and σ_2', and label stacks $\mathcal{M}'$ and L' such that

- $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \longrightarrow^{+} \langle s_1' \mid (CT, \sigma_1', \mathcal{M}', L')\rangle$,
- $\langle s_2 \mid (CT, \sigma_2, \mathcal{M}, L)\rangle \xrightarrow{\bullet} \langle s_2' \mid (CT, \sigma_2', \mathcal{M}', L')\rangle$,
- *s_1' and s_2' are endorsement-free at ℓ_t,*
- $\vdash \sigma_i'$ wt *for both $i = 1, 2$,*
- $\sigma_1' \approx_{\ell_t} \sigma_2'$ *with* $\sigma_2' \subseteq \sigma_1'|^{\bullet}_{\ell_t}$, *and*
- $\Sigma_{\sigma_1'}; \Gamma; pc; \ell_I \vdash s_1' : \tau \dashv \ell_O \geq_{\bullet} \Sigma_{\sigma_2'}; \Gamma; pc; \ell_I \vdash s_2' : \tau \dashv \ell_O$.

Proof. By Lemma 4.10, the fact that $s_1 \longrightarrow^{+} v$ means that $s_2 \xrightarrow{\bullet} s_2'$. This will be a proof by induction on the rule used to prove $s_2 \xrightarrow{\bullet} s_2'$, though in the case of B-TREF, we may need to construct a new, different s_2'.

Case B-PURESTEP**:** We have that $s_1 \longrightarrow s_1'$ by whatever step was used in the hypothesis of B-PureStep. To prove the typing proofs correspond, we note that, for most possible steps, both s'_1 and s'_2 type-check by the same logic as in the proof of Theorem 4.6, meaning the typing proofs transform in the same way. The exception is E-Endorse. Here let $s_2 = \mathsf{endorse}\ v_2\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell$ and consider two cases: if $v_2 = \bullet$ and if $v_2 \neq \bullet$. When $v_2 \neq \bullet$, the same argument as in Theorem 4.6 applies, and $s_1 = \mathsf{endorse}\ v_1\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell$ follows the same step by the same argument. When $v_2 = \bullet$, inversion on the typing rules gives us that $\ell' \not\Rightarrow \ell_t$. Because we know s_2 is endorsement-free at ℓ_t, this means $\ell \not\Rightarrow \ell_t$, so therefore $\Sigma; \Gamma \vdash \bullet : t^{\ell}$. Again, s_1 follows E-Endorse and the typing proofs correspond.

For the heap correspondence and well-typed conditions, we note that the heaps and their types remain unchanged for both executions. To maintain endorsement-freedom at ℓ_t, most possible steps cannot add new terms, so they cannot add new endorse terms. E-Call and E-CallAtk, however, can introduce new terms into s'_1 and s'_2 that may not have been present in s_1 and s_2. Because CT is endorsement-free at ℓ_t, any new sub-statements of the form $\mathsf{endorse}\ v\ \mathsf{from}\ \ell'\ \mathsf{to}\ \ell$ must have the required property.

Case B-Eval**:** In this case $s_2 = E_2[\tilde{s}_2]$ and $\tilde{s}_2 \multimap\!\!\!\rightarrow \tilde{s}'_2$. By inversion on $s_1 \geq_\bullet s_2$, it must be the case that $s_1 = E_1[\tilde{s}_1]$ where $\tilde{s}_1 \geq_\bullet \tilde{s}_2$. By inversion on the set of evaluation contexts, s_1 can only step through E-Eval and no other steps. Therefore, by induction, on the $\multimap\!\!\!\rightarrow$ relation, $\tilde{s}_1 \longrightarrow \tilde{s}'_1$ with the required properties, so E-Eval gives us everything except correspondence of the typing proof. We get that by noting that we can apply Lemma 4.5 in exactly the same way to both proofs.

Case B-BulletCtx **with** $B = ![\cdot]$**,** $(C)[\cdot]$**, or** $[\cdot].f$**:** Here we have that $s_2 = B[\bullet]$, so by inversion on $s_1 \geq_\bullet s_2$, we know that $s_1 = B[v_1]$ for some non-bullet value

v_1. By the fact that $s_1 \longrightarrow^+ v$, we know that s_1 must step, so by inspection on the operational semantics, it must step with E-DEREF, E-CAST, or E-FIELD, depending on the syntactic form. In each case the result is a non-variable value v_1', so therefore $s_1' = v_1' \geq_\bullet \bullet = v_2'$ with typing proofs using VAL to get to a value typing judgment that allows them to differ on $\bullet$. The heap does not change.

Case B-BULLETCTX **with** $B = \mathsf{if}\{pc'\}\ [\cdot]\ \mathsf{then}\ e_1^2\ \mathsf{else}\ e_2^2$**:** First we note that $s_2' = \bullet$ and $\sigma_2' = \sigma_2$. Inversion on the typing rules proves that $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \mathsf{bool}^\ell$ for some $\ell \not\Rightarrow \ell_t$ and $\ell \triangleleft \tau$, meaning BULLET gives us $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \tau$. We now examine s_1 and the corresponding steps.

Because $\Sigma_{\sigma_1}; \Gamma; pc; \ell_\mathrm{I} \vdash s_1 : \tau \dashv \ell_\mathrm{O} \geq_\bullet \Sigma_{\sigma_2}; \Gamma; pc; \ell_\mathrm{I} \vdash s_2 : \tau \dashv \ell_\mathrm{O}$, it must be the case that $s_1 = \mathsf{if}\{pc'\}\ v_1\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_2^1$. This syntactic structure ensures that s_1 must step with one of E-IFT or E-IFF. Because we have assumed that $\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \longrightarrow^+ \langle v \mid C\rangle$, we further know that

$$\begin{aligned}\langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle &\longrightarrow \langle e_i^1\ \mathsf{at\text{-}pc}\ pc' \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \\ &\longrightarrow^* \langle v\ \mathsf{at\text{-}pc}\ pc' \mid C\rangle \\ &\longrightarrow \langle v \mid C\rangle.\end{aligned}$$

By inspection on the semantic rules, E-EVAL must apply in each of the steps in the middle segment, meaning $\langle e_i^1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid C\rangle$.

The correspondence of the typing proof with s_1 proves that $\Sigma_{\sigma_1}; \Gamma \vdash v_1 : \mathsf{bool}^\ell$ for some $\ell \not\Rightarrow \ell_t$. Inversion on that typing proof reveals that $\ell \Rightarrow pc'$ and $\Sigma_{\sigma_1}; \Gamma; pc'; \ell_\mathrm{I} \vdash e_i^1 : \tau \dashv \ell_\mathrm{O}$. By Corollary 4.1, $\mathcal{M}' = \mathcal{M}$ and $L' = L$, and $\sigma_1' \approx_{\ell_t} \sigma_1 \approx_{\ell_t} \sigma_2 = \sigma_2'$. By inductively applying Theorem 4.6, we get $\sigma_1' \supseteq \sigma_1$, so

$$\sigma_2' = \sigma_2 \subseteq \sigma_1|^\bullet_{\ell_t} \subseteq \sigma_1'|^\bullet_{\ell_t}.$$

By letting $s'_1 = v$ and noting that all values are endorsement-free at ℓ_t, we complete the case.

Case B-BULLETCTX **with** $[\cdot].m(\overline{v})$**:** This case is very similar to the previous case. Again, $s'_2 = \bullet$ and $\sigma'_2 = \sigma_2$. Also, inversion on the typing rules gives us $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : C^{\ell}$ for some $\ell \not\Rightarrow \ell_t$, and $\ell \triangleleft \tau$, again allowing BULLET to prove $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \tau$. We again turn to s_1.

The typing correspondence now means $s_1 = v_1.m(\overline{w})$, so it must step using E-CALL or E-CALLATK. Therefore $v_1 = \mathsf{new}\ C(\overline{w'})$ and $mbody(C, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)$. Again, we know that it steps to a value, so now

$$\begin{aligned} \langle s_1 \mid (CT, \sigma_1, \mathcal{M}, L)\rangle &\longrightarrow \langle(\mathsf{return}_{\tau}\ e')\ \mathsf{at\text{-}pc}\ pc_2 \mid (CT, \sigma_1, (\mathcal{M}, \ell_m), L)\rangle \\ &\longrightarrow^{*} \langle(\mathsf{return}_{\tau}\ v)\ \mathsf{at\text{-}pc}\ pc_2 \mid (CT, \sigma'_1, \mathcal{M}'_1, L'_1)\rangle \end{aligned}$$

where $e' = e[\overline{x} \mapsto \overline{w}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{w'})]$ is an expression. Additionally, Theorem 4.6 ensures $\Sigma_{\sigma_1}; \Gamma; pc_2; \ell'_{\mathrm{I}} \vdash e' : \tau \dashv \ell'_{\mathrm{O}}$. By the correspondence of the typing proofs of s_1 and s_2, we know that $\Sigma_{\sigma_2}; \Gamma; pc; \ell_{\mathrm{I}} \vdash \bullet.m(\overline{v}) : \tau \dashv \ell_{\mathrm{O}}$ interpreting $\bullet$ as $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : C^{\ell}$. Inversion on the typing rules therefore gives us that $\ell \not\Rightarrow \ell_t$ and $\ell \Rightarrow pc_1$. By transitivity, we know that $pc_1 \not\Rightarrow \ell_t$, so by the fact that CT is endorsement-free at ℓ_t, we have that $pc_2 \not\Rightarrow \ell_t$. Therefore, we can apply Corollary 4.1 to our above semantic steps, giving:

- $\sigma'_1 \approx_{\ell_t} \sigma_1 \approx_{\ell_t} \sigma_2 = \sigma'_2$,
- $\mathcal{M}'_1 = \mathcal{M}, \ell_m$, and
- $L'_1 = L$.

Again, Theorem 4.6's result tells us $\sigma'_1 \supseteq \sigma_1$, meaning $\sigma'_2 \subseteq \sigma'_1|^{\bullet}_{\ell_t}$. Applying E-ATPC and E-RETURN while letting $s'_1 = v$ completes the case.

Case B-TREF**:** Here we take the $\tilde{s}'_2$ from Lemma 4.10 as a candidate, which we may modify. In particular, we note that $s_2 = \mathsf{ref}\ v_2\ \tau'$, so by the typing correspondence, $s_1 = \text{ref } v_1\ \tau'$. Therefore, s_1 must step using E-REF, giving $s'_1 = \iota$ for some $\iota \notin \text{dom}(\sigma_1)$ and $\sigma'_1 = \sigma_1[\iota \mapsto (v_1, \tau')]$.

For s'_2, we know that $\text{dom}(\sigma_2) \subseteq \text{dom}(\sigma_1)$, so $\iota \notin \text{dom}(\sigma_2)$. We also know that s_2 could step using B-TREF, so we can use the same step, setting the location to ι. Therefore, $\sigma'_2 = \sigma_2[\iota \mapsto (v_2, \tau')]$. The fact that s'_2 is well-typed in $\Sigma_{\sigma'_2}$ follows directly from this extension.

Endorsement-freedom of s'_1 and s'_2, typing correspondence, and that $\sigma'_1 \approx_{\ell_t} \sigma'_2$ are now straightforward. To show that $\sigma'_2 \subseteq \sigma'_1|^{\bullet}_{\ell_t}$, we note that the typing correspondence between v_1 and v_2 means that either $v_2 = v_1$ or $v_2 = \bullet$. The second case is impossible because $\text{label}(\tau') \Rightarrow \ell_t$, so inversion on the typing rules demonstrates $\Sigma_{\sigma_2}; \Gamma \nvdash \bullet : \tau'$. With $v_2 = v_1$, the relation between σ'_1 and σ'_2 follows directly from their definitions and the corresponding relation between σ_1 and σ_2.

Case B-UREF: Using the same logic as the previous case, s_1 must step using E-REF, and we can make $s'_2 = s'_1 = \iota$ for some $\iota \notin \text{dom}(\sigma_1) \supseteq \text{dom}(\sigma_2)$. We again have that s'_1 and s'_2 correspond and are well-typed and that $\sigma'_1 \approx_{\ell_t} \sigma'_2$. Finally, we note that, by assumption from B-UREF, $s_2 = \text{ref } v_2\ \tau'$ where $\text{label}(\tau') \not\Rightarrow \ell_t$. Therefore, $\sigma'_1|^{\bullet}_{\ell_t} = \sigma_1[\iota \mapsto \tau']]|^{\bullet}_{\ell_t} = \sigma_1|^{\bullet}_{\ell_t}[\iota \mapsto (\bullet, \tau')]$, and correspondingly, $\sigma'_2 = \sigma_2[\iota \mapsto (\bullet, \tau')]$. The correspondence follows from the correspondence between σ_1 and σ_2.

Case B-BASSIGN: Here $s_2 = \bullet := v_2$, so $s_1 = \iota := v_1$ where $v_1 \geq_\bullet v_2$. By inversion on the typing rules and the $\geq_\bullet$ relation, we know that $\Sigma_{\sigma_1}; \Gamma \vdash \iota : \text{ref } \tau'^{\ell}$ and $\Sigma_{\sigma_2}; \Gamma \vdash \bullet : \text{ref } \tau'^{\ell}$. Moreover, we know that $\ell \not\Rightarrow \ell_t$ and $\ell \triangleleft \tau$. Since s_1 steps, it must step with E-ASSIGN, meaning $\sigma'_1 = \sigma_1[\iota \mapsto (v_1, \tau')]$. Therefore $\sigma'_1|^{\bullet}_{\ell_t} = \sigma_1|^{\bullet}_{\ell_t} \supseteq \sigma_2 = \sigma'_2$. Letting $s'_1 = s'_2 = ()$ completes the case.

Case B-TASSIGN: This is similar to the B-TREF case, but we do not need to construct a new location, as $s_2 = \iota := v$. We also know by the same logic as in that case that $v \neq \bullet$, so $s_1 = s_2$. B-TASSIGN and E-ASSIGN produce precisely the same output on the same input, proving the case.

Case B-UASSIGN**:** Here we note that $s_2 = \iota := v_2$, meaning $s_1 = \iota := v_1$ where $v_1 \geq_\bullet v_2$ and $i \in \mathrm{dom}(\sigma_2)$. By inversion on the typing rules, we know that $\Sigma_{\sigma_i}(\iota) = \tau'$ for both $i = 1, 2$ and some τ' where $\mathrm{label}(\tau') \not\Rightarrow \ell_t$. Therefore, s_1 steps using E-ASSIGN, so $\sigma_1' = \sigma_1[\iota \mapsto (v_1, \tau')]$ where $\sigma_1(\iota) = (v, \tau')$ for some v. As a result, $\sigma_2' = \sigma_2 \subseteq \sigma_1|^\bullet_{\ell_t} = \sigma_1'|^\bullet_{\ell_t}$. The two steps result in $s_1' = s_2' = ()$, so the statements type-check with corresponding rules. □

Corollary 4.2. *For any class table CT, heap type Σ, and expression (not statement) e, if*

- $\Sigma \vdash CT$ ok *is endorsement-free at ℓ_t,*
- *e is endorsement-free at ℓ_t,*
- $\Sigma \subseteq \Sigma_{\sigma_1}$,
- $\Sigma; \Gamma; pc; \ell_I \vdash e : \tau \dashv \ell_O$,
- $\vdash \sigma_1$ wt, *and*
- $\langle e \mid (CT, \sigma_1, \mathcal{M}, L)\rangle \longrightarrow^* \langle v \mid (CT, \sigma_1', \mathcal{M}', L')\rangle$,

then there is some value v', heap σ_2', and heap type Σ_2' such that

- $\langle e \mid (CT, \sigma_1|^\bullet_{\ell_t}, \mathcal{M}, L)\rangle \multimap\!\!\!\bullet\!\!\to^* \langle v' \mid (CT, \sigma_2', \mathcal{M}', L')\rangle$ *and*
- $\sigma_1' \simeq_{\ell_t} \sigma_2'$.

Proof. This proof follows from Lemma 4.11 and induction on the number of steps, letting $s_1 = s_2 = e$ and $\sigma_2 = \sigma_1|^\bullet_{\ell_t}$ to start. If there are zero steps—that is $e = s_1 = v$—then we are done. Otherwise Lemma 4.11 allows us to step s_2 once using $\multimap\!\!\!\bullet\!\!\to$ and provides a corresponding set of steps using $\longrightarrow$ for s_1. The result may have differently-named locations from the original, but Theorem 4.5 allows us to continue stepping a location-name isomorphic expression. The steps therefore maintain all requirements to apply Lemma 4.11 again until s_1 reaches a value. At that point, we are assured $\sigma_2' \subseteq \sigma_1''|^{\bullet}_{\ell_t}$ and $\sigma_1' \approx_{\ell_t} \sigma_1''$ for some $\sigma_1'' \simeq \sigma_1'$. Therefore $\sigma_1' \simeq_{\ell_t} \sigma_2'$. □

Theorem 4.1 (Noninterference). *Let CT be a class table where $\Sigma \vdash CT$ ok is endorsement-free at ℓ_t. For any well-typed heaps σ_1 and σ_2 such that $\Sigma \subseteq \Sigma_{\sigma_i}$ and any invocation I such that $\Sigma \vdash I$ and $(I, CT, \sigma_i) \Downarrow \sigma_i'$, if $\sigma_1 \simeq_{\ell_t} \sigma_2$, then $\sigma_1' \simeq_{\ell_t} \sigma_2'$.*

Proof. First we note that since $\Sigma \subseteq \Sigma_{\sigma_i}$, Lemma 4.4 means $\Sigma_{\sigma_i} \vdash CT$ ok for both $i = 1, 2$, meaning our various lemmas apply in both cases. Without loss of generality, we assume $\sigma_1 \approx_{\ell_t} \sigma_2$, since we can permute the location names in one to match the other and permute the results back later. There exists a unique

$$\tilde{\sigma} = \sigma_1|^{\bullet}_{\ell_t} = \sigma_2|^{\bullet}_{\ell_t}$$

Let $I = (\iota, m(\overline{v}), \ell)$. Note that $!\iota.m(\overline{v})$ is an expression with no $\mathsf{endorse}$ statements and $\Sigma \vdash\, !\iota.m(\overline{v}) : \tau$ for some τ. Therefore, by Corollary 4.2,

$$\langle !\iota.m(\overline{v}) \mid (CT, \tilde{\sigma}, \ell, \cdot)\rangle \longrightarrow^{*} \langle v \mid (CT, \tilde{\sigma}', \ell, \cdot)\rangle$$

where $\tilde{\sigma}' \simeq_{\ell_t} \sigma_i'$ for both $i = 1, 2$. Transitivity of $\simeq_{\ell_t}$ then proves $\sigma_1' \simeq_{\ell_t} \sigma_2'$. □

4.12 Proof of Reentrancy Security

We now prove Theorem 4.2. As discussed in Section 4.5.3, we do this by first proving Theorem 4.3 saying all reentrancy is tail-reentrancy and Theorem 4.4 that says tail reentrancy is secure according on Definition 4.9.

4.12.1 SeRIF Allows Only Tail Reentrancy

We start by proving Theorem 4.3. We prove this theorem using the general formulation of "trusted" and "untrusted" labels. In particular, we partition $\mathcal{L}$ into a downward-closed sublattice $\mathcal{T}$ and the attacker-controlled labels $\mathcal{A} = \overline{\mathcal{T}}$. Notationally, we will use ℓ_t to denote some trusted label ($\ell_t \in \mathcal{T}$), rather than a distinguished one. We refer to code complying with locks in $\mathcal{T}$-code, to mean it complies with locks in ℓ_t-code for all $\ell_t \in \mathcal{T}$.

Finally, we will prove the result for two adversarial models: one in which the E-CallAtk rule is admissible and $\mathcal{A}$ is a sublattice, and the other where the E-CallAtk rule is *not* admissible, but $\mathcal{A}$ has no restrictions beyond $\mathcal{A} = \overline{\mathcal{T}}$. These two proofs are extremely similar. Indeed, they differ only in a single case of Lemma 4.15 and Lemma 4.16 on which it relies. We will specifically call out the differences when they arise.

The proof follows the following general structure. First we show that high-integrity code maintains all of the input locks ℓ_I it claims to and the operational semantics maintain all dynamic locks. Second, we will show that, if a statement that complies with locks steps to an auto-endorse call, it cannot comply with a lock on any label that call endorses through (i.e., one that does not trust pc_1 but does trust pc_2). Finally, we connect these to show that, for low-integrity call from a high-integrity context that proceeds to make a reentrant call, the original low-integrity call must have been in tail position form the original high-integrity execution.

To discuss the security of an invocation mid-evaluation, we need to discuss the security of a statement s with respect to locks. We do this using several different tools. First, we extend our notion of lock compliance in $\mathcal{T}$-code to statements. We do this with a judgment $pc \vdash_{\mathcal{T}} s$ cwl. The nontrivial rules are as follows.

$$\frac{pc \vdash_{\mathcal{T}} e_1 \text{ cwl} \qquad pc \vdash_{\mathcal{T}} e_2 \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{if}\{pc'\}\ v \text{ then } e_1 \text{ else } e_2) \text{ cwl}} \qquad \frac{pc \vdash_{\mathcal{T}} e \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{lock } \ell \text{ in } e) \text{ cwl}}$$

$$\frac{pc \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (s \text{ with-lock } \ell) \text{ cwl}} \qquad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl} \qquad pc \vdash_{\mathcal{T}} e \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{let } x = s \text{ in } e) \text{ cwl}} \qquad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (\text{return}_{\tau}\ s) \text{ cwl}}$$

$$\frac{pc' \vdash_{\mathcal{T}} s \text{ cwl}}{pc \vdash_{\mathcal{T}} (s \text{ at-pc } pc') \text{ cwl}} \qquad \frac{pc \vdash_{\mathcal{T}} s \text{ cwl} \qquad pc \notin \mathcal{T}}{pc \vdash_{\mathcal{T}} (\text{ignore-locks-in } s) \text{ cwl}}$$

If s has none of the syntactic forms in the rules defined above, then $pc \vdash_{\mathcal{T}} s$ cwl for any pc and $\mathcal{T}$. Note that, because at-pc terms are statements but not expressions, for any expression e, $pc \vdash_{\mathcal{T}} e$ cwl if one of two conditions holds: either $pc \notin \mathcal{T}$ or e has no subexpressions of the form ignore-locks-in e'. As a result, we can also specify our definition of lock compliance from class tables using this judgment. Specifically,

$$\begin{gathered} CT \text{ complies with locks in } \mathcal{T}\text{-code} \\ \Updownarrow \\ \dfrac{\begin{gathered} CT(C) = \text{class } C[\ell_C] \text{ extends } D\ \{\overline{f} : \overline{\tau_f}\ ;\ K\ ;\ \overline{M}\} \\ \tau\ m\{pc_1 \gg pc_2; \ell_O\}(\overline{x} : \overline{\tau_a})\ \{e\} \in \overline{M} \end{gathered}}{\ell_C \vdash_{\mathcal{T}} e \text{ cwl}} \text{ is admissible for } CT \end{gathered}$$

Lemma 4.12. *If $pc \vdash_{\mathcal{T}} s$ cwl and $pc \Rightarrow pc'$, then $pc' \vdash_{\mathcal{T}} s$ cwl.*

Proof. By simple induction on the definition of $pc \vdash_{\mathcal{T}} s$ cwl. □

We will also be considering statements in the middle of evaluation, so we need a way to extract the pc label that we expect sub-statements to type-check with, and similarly we need to extract the list of dynamic locks that will be present when a sub-statement completes executing. We do that using the following two recursive functions defined on evaluation contexts.

$$getLocks(L, E) = \begin{cases} L & \text{if } E = [\cdot] \\ getLocks((L, \ell), E') & \text{if } E = E' \ \textsf{with-lock}\ \ell \\ getLocks(L, E') & \text{if } E = \textsf{let}\ x = E'\ \textsf{in}\ e,\ \textsf{return}_\tau\ E',\ \text{or } E'\ \textsf{at-pc}\ pc \end{cases}$$

$$innerPc(pc, E) = \begin{cases} pc & \text{if } E = [\cdot] \\ innerPc(pc', E') & \text{if } E = E' \ \textsf{at-pc}\ pc' \\ innerPc(pc, E') & \text{if } E = \textsf{let}\ x = E'\ \textsf{in}\ e,\ \textsf{return}_\tau\ E',\ \text{or } E'\ \textsf{with-lock}\ \ell \end{cases}$$

We extend both of these to statements by $getLocks(L, E[e]) = getLocks(L, E)$, and similarly for *innerPc*.

Definition 4.12 (Configuration Safety). A pair $\langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle$ of statement and configuration is *$\mathcal{T}$-safe with pc and $\hat{L}$* if

1. $\Sigma_\sigma \vdash CT$ ok complies with locks in $\mathcal{T}$-code,
2. $\vdash \sigma$ wt,
3. $\Sigma_\sigma; \Gamma; pc; \ell_\mathrm{I} \vdash s : \tau \dashv \ell_\mathrm{O}$,
4. $pc \vdash_\mathcal{T} s$ cwl,
5. $L = getLocks(\hat{L}, s)$, and
6. for any E and s' where $E \neq E'[\textsf{return}_\tau\ [\cdot]]$ and $pc' = innerPc(pc, E) \in \mathcal{T}$, if $s = E[s']$ then there is some ℓ'_I such that $\Sigma_\sigma; \Gamma'; pc'; \ell'_\mathrm{I} \vdash s' : \tau' \dashv \ell'_\mathrm{O}$ and $(\bigwedge getLocks(\hat{L}, E)) \wedge \ell'_\mathrm{I} \Rightarrow pc'$.

Lemma 4.13. *If $\langle E[s] \mid C\rangle$ is $\mathcal{T}$-safe at pc and L, then $\langle s \mid C\rangle$ is $\mathcal{T}$-safe at innerPc(pc, E) and getLocks(L, E).*

Proof. By induction on E and the definitions of *innerPc* and *getLocks*. □

Lemma 4.14 (Preservation of $\mathcal{T}$-Safety). *If $\langle s \mid C\rangle$ is $\mathcal{T}$-safe with pc and $\hat{L}$, and $\langle s \mid C\rangle \longrightarrow \langle s' \mid C'\rangle$, then $\langle s' \mid C'\rangle$ is $\mathcal{T}$-safe with pc and $\hat{L}$.*

Proof. Condition 1 follows from Lemma 4.3 and the fact that CT must remain unchanged. Conditions 2 and 3 follow directly from Theorem 4.6. We prove the other three conditions by induction on the operational semantics. Notationally, we let $C = (CT, \sigma, \mathcal{M}, L)$ and $C' = (CT, \sigma', \mathcal{M}', L')$. Also, by assumption, there is some ℓ'_{I} such that $\Sigma_\sigma; \Gamma; pc; \ell'_{\mathrm{I}} \vdash s : \tau \dashv \ell_{\mathrm{O}}$ and $(\bigwedge \hat{L}) \wedge \ell'_{\mathrm{I}} \Rightarrow pc$. We assume without loss of generality that ℓ_{I} has this property.

Case E-EVAL**:** In this case $s = E[\tilde{s}]$, $\langle \tilde{s} \mid C\rangle \longrightarrow \langle \tilde{s}' \mid C'\rangle$, and $s' = E[\tilde{s}']$. Let $pc' = \mathit{innerPc}(pc, E)$ and $\hat{L}' = \mathit{getLocks}(\hat{L}, E)$. By Lemma 4.13, we know that $\tilde{s}$ is $\mathcal{T}$-safe at pc' and $\hat{L}'$, so by induction on the operational semantics, $\langle \tilde{s}' \mid C'\rangle$ is as well. We also note that $\Sigma_\sigma; \Gamma'; pc'; \ell'_{\mathrm{I}} \vdash \tilde{s} : \tau' \dashv \ell'_{\mathrm{O}}$.

By the safety of $\langle s \mid C\rangle$, for every pair of sub-contexts E_1 and E_2 such that $E = E_1[E_2]$, either $\tilde{pc} = \mathit{innerPc}(pc, E_1) \notin T$, $E_1 = E'_1[\mathsf{return}_\tau\ [\cdot]]$, or $\Sigma_\sigma; \tilde{\Gamma}; \tilde{pc}; \tilde{\ell}_{\mathrm{I}} \vdash E_2[\tilde{s}] : \tau_1 \dashv \tilde{\ell}_{\mathrm{O}}$ for some $\tilde{\ell}_{\mathrm{I}}$ where $(\bigwedge \mathit{getLocks}(\hat{L}, E_1)) \wedge \tilde{\ell}_{\mathrm{I}} \Rightarrow \tilde{pc}$. By Theorem 4.6, $\Sigma_{\sigma'}; \Gamma'; pc'; \ell'_{\mathrm{I}} \vdash \tilde{s}' : \tau' \dashv \ell'_{\mathrm{O}}$, so by Lemma 4.5, we also have that $\Sigma_{\sigma'}; \tilde{\Gamma}; \tilde{pc}; \tilde{\ell}_{\mathrm{I}} \vdash E_2[\tilde{s}'] : \tau_1 \dashv \tilde{\ell}_{\mathrm{O}}$. As this holds for every choice of E_1 and E_2, this proves the case.

Cases E-IFT **and** E-IFF**:** In both cases we have $s = \mathsf{if}\{pc'\}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ and $s' = e_i\ \mathsf{at\text{-}pc}\ pc'$ for either $i = 1$ or 2. By inversion on the typing rules, $pc \Rightarrow pc'$ and $\Sigma_\sigma; \Gamma; pc'; \ell_{\mathrm{I}} \vdash e_i : \tau \dashv \ell_{\mathrm{O}}$ for both $i = 1, 2$. Moreover, by Lemma 4.12, $pc' \vdash_{\mathcal{T}} e_i\ \mathsf{cwl}$, so Condition 4 holds for s'. Because e_1 and e_2 are expressions, we know that if $e_i = E[\tilde{s}]$, then E consists entirely of let and $\mathsf{ignore\text{-}locks\text{-}in}$ statements and $\tilde{s}$ is an expression. Therefore, if $pc' \notin T$, then Condition 6 is trivial.

If $pc' \in T$, then because $\mathcal{T}$ is downward-closed, $pc \in T$. Because $pc \vdash_{\mathcal{T}} e_i\ \mathsf{cwl}$, $\mathsf{ignore\text{-}locks\text{-}in}$ cannot appear in e in this sub-case, so E consists entirely of let statements. As a result, $\Sigma_\sigma; \Gamma'; pc'; \ell_{\mathrm{I}} \vdash \tilde{s} : \tau' \dashv \ell_{\mathrm{O}}$ for some $\Gamma' \supseteq \Gamma$ and τ'.

Because $(\bigwedge \hat{L}) \wedge \ell_I \Rightarrow pc \Rightarrow pc'$, this proves that $\langle e_i \mid C' \rangle$ is $\mathcal{T}$-safe at pc' and $\hat{L}$. Since $s' = e_i$ $\mathsf{at\text{-}pc}$ pc', the $\mathcal{T}$-safety transfers to $\langle s' \mid C' \rangle$.

Case E-LET**:** Here $s = (\mathsf{let}\ x = v\ \mathsf{in}\ e)$ and $s' = e[x \mapsto v]$. Theorem 4.6 proves $\Sigma_\sigma; \Gamma; pc; \ell_I \vdash s' : \tau \dashv \ell_O$. Moreover, because s' is an expression, by the same logic as in the previous case, $\langle s' \mid C' \rangle$ must be $\mathcal{T}$-safe at pc and $\hat{L}$.

Case E-LOCK**:** In this case $s = \mathsf{lock}\ \ell\ \mathsf{in}\ e$. To prove Condition 5 holds, simply note that $L' = (L, \ell) = (\hat{L}, \ell) = \mathit{getLocks}(\hat{L}, e\ \mathsf{with\text{-}lock}\ \ell)$.

Next, inversion on the typing rules tells that $\Sigma_\sigma; \Gamma; pc; \ell'_I \vdash e : \tau \dashv \ell'_O$ where $\ell'_I \wedge \ell \Rightarrow \ell_I$ and $\ell'_O \wedge \ell \Rightarrow \ell_O$. Further, we know that $s' = e\ \mathsf{with\text{-}lock}\ \ell$, $L = \hat{L}$, and $L' = (L, \ell)$ By Condition 4 on s, first Condition 4 holds trivially on s', and second, either $pc \in T$ or e contains no $\mathsf{ignore\text{-}locks\text{-}in}$ terms, as e is an expression. Therefore, by the same logic as in the previous two cases, it suffices to show Condition 6 holds when $pc \in T$ and $E = [\cdot]\ \mathsf{with\text{-}lock}\ \ell$. Here we know that $\Sigma_\sigma; \Gamma; pc; \ell'_I \vdash e : \tau \dashv \ell'_O$ with ℓ'_I defined as above. As a result,

$$\begin{aligned}
\left(\bigwedge \mathit{getLocks}(\hat{L}, E)\right) \wedge \ell'_I &= \left(\bigwedge (\hat{L}, \ell)\right) \wedge \ell'_I \\
&= \left(\bigwedge \hat{L}\right) \wedge \ell \wedge \ell'_I \\
&\Rightarrow \left(\bigwedge \hat{L}\right) \wedge \ell_I \\
&\Rightarrow pc.
\end{aligned}$$

Case E-UNLOCK**:** Here $s = v\ \mathsf{with\text{-}lock}\ \ell$ and $s' = v$, so Condition 4 is trivial. Condition 5 follows from the semantic rule that requires $L = (L', \ell)$, so if $\mathit{getLocks}(\hat{L}, s) = L$, then $\hat{L} = L' = \mathit{getLocks}(\hat{L}, v)$. Condition 6 follows from the fact that values type-check with any ℓ_I, including pc.

Cases E-CALL **and** E-CALLATK**:** In both of these cases, $s = \mathsf{new}\ C(\overline{v}).m(\overline{w})$. If we let $\mathit{mbody}(C, m) = (\ell_m, \overline{x}, \overline{\tau_a}, pc_1 \gg pc_2, e, \tau)$, then $s' = (\mathsf{return}_\tau\ e')\ \mathsf{at\text{-}pc}\ pc_2$ where

$e' = e[\overline{x} \mapsto \overline{w}, \textsf{this} \mapsto \textsf{new } C(\overline{v})]$. By Condition 1 on s, we know that $\ell_C \Vdash_{\mathcal{T}} e'$ $\textsf{cwl}$. The METHOD-OK rule requires that $\ell_C \Rightarrow pc_2$, so therefore by Lemma 4.12 proves $pc_2 \Vdash_{\mathcal{T}} e'$ $\textsf{cwl}$, proving Condition 4.

Since the body of the method is an expression and $L = L'$, Condition 5 holds trivially.

For Condition 6, we consider multiple possible evaluation contexts E. If $E = [\cdot]$, note that $\textsf{return}_\tau\ s''$ type-checks with any ℓ_I. If $E = \textsf{return}_\tau\ [\cdot]$, then this is precisely the caveat that Condition 6 does not restrict. If $E = \textsf{return}_\tau\ [\cdot]\ \textsf{at-pc}\ pc_2$, METHOD-OK ensures that $\Sigma_\sigma; \cdot; pc_2; \ell'_I \vdash e' : \tau \dashv \ell'_O$ for some $\ell'_I \Rightarrow pc_2$. In particular, this means $(\bigwedge \hat{L}) \wedge \ell'_I \Rightarrow pc_2$ regardless of the contents of $\hat{L}$. Moreover, because we know that $pc_2 \Vdash_{\mathcal{T}} e'$ $\textsf{cwl}$ and e' is an expression, either $pc_2 \notin \mathcal{T}$, in which case Condition 6 is trivial in e', or $pc_2 \in \mathcal{T}$ and e' does not contain $\textsf{ignore-locks-in}$ terms. In the second case, the same logic as in several previous cases completes the proof that Condition 6 holds, and thus the case.

In all other cases the step leaves L unchanged and produces a value. All well-typed value type check with any ℓ_I and $pc \Vdash_{\mathcal{T}} v$ $\textsf{cwl}$ for any label pc and value v, so all conditions hold. □

Lemma 4.15. *For any label $\ell_t \in \mathcal{T}$, statement s, configuration $C = (CT, \sigma, \mathcal{M}, L)$, lock list $\hat{L}$, if*

1. *$\langle s \mid C \rangle$ is $\mathcal{T}$-safe with pc and $\hat{L}$ for some label pc,*
2. *s contains no sub-statements of the form* $\textsf{ignore-locks-in}$ *s',*
3. *$\langle s \mid C \rangle \longrightarrow^* \langle E[\textsf{new } C(\overline{v}).m(\overline{w})] \mid C' \rangle \longrightarrow \langle s' \mid C'' \rangle$, and*
4. *$mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \hat{\tau}$ with $pc_1 \not\Rightarrow \ell_t$ and $pc_2 \Rightarrow \ell_t$,*

then for any ℓ_I and ℓ_O such that $\Sigma_\sigma; \Gamma; pc; \ell_I \vdash s : \tau \dashv \ell_O$, then $(\bigwedge \hat{L}) \wedge (\ell_I \vee \ell_O) \not\Rightarrow \ell_t$.

Proof. This is a proof by induction on the number of steps in premise 3. For the base case of zero steps, $s = E[\textsf{new } C(\overline{v}).m(\overline{w})]$. We prove this case by induction on E.

For these cases, we will use the notational short-hand $s' = E'[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ where E' will be defined in each inductive case.

Case $E = [\cdot]$**:** The fact that $\langle s \mid C\rangle$ is $\mathcal{T}$-safe with pc and $\hat{L}$ directly means that $\Sigma_\sigma; \Gamma; pc; \ell_\mathrm{I} \vdash \mathsf{new}\ C(\overline{v}).m(\overline{w}) : \tau \dashv \ell_\mathrm{O}$. Inversion on the typing rules proves that $pc_1 \Rightarrow pc_2 \vee \ell_\mathrm{I}$. Because this expression steps again, inversion on the operational semantics ensures that it must step using E-CALL, and therefore $\bigwedge_{\ell \in L}(pc_1 \Rightarrow pc_2 \vee \ell)$. Because $\wedge$ produces the greatest lower bound and the lattice is distributive,

$$pc_1 \Rightarrow \bigwedge_{\ell \in L}(pc_2 \vee \ell) \wedge (pc_2 \vee \ell_\mathrm{I}) = pc_2 \vee \Big(\big(\bigwedge L\big) \wedge \ell_\mathrm{I}\Big).$$

Moreover, because $pc_1 \not\Rightarrow \ell_t$, transitivity of $\Rightarrow$ tells us that this label does not act for ℓ_t. Yet $pc_2 \Rightarrow \ell_t$, so by the definition of join, it must be the case that $((\bigwedge L) \wedge \ell_\mathrm{I}) \not\Rightarrow \ell_t$. Because $\ell_\mathrm{I} \Rightarrow \ell_\mathrm{I} \vee \ell_\mathrm{O}$, and $L = \hat{L}$ in this case, transitivity of $\Rightarrow$ and equality substitution proves $(\bigwedge \hat{L}) \wedge (\ell_\mathrm{I} \vee \ell_\mathrm{O}) \not\Rightarrow \ell_t$, as desired.

Case $E = \mathsf{let}\ x = E'\ \mathsf{in}\ e$**:** In this case, inversion on the typing rules guarantees that $\Sigma_\sigma; \Gamma; pc; \ell_\mathrm{I} \vdash s' : \tau' \dashv \ell'_\mathrm{O}$ where $\ell'_\mathrm{O} \Rightarrow \ell_\mathrm{I}$. Premises 1 and 2 are clearly true for $\langle s' \mid C\rangle$, so by induction on E, $(\bigwedge \hat{L}) \wedge (\ell_\mathrm{I} \vee \ell'_\mathrm{O}) \not\Rightarrow \ell_t$. Since $\ell'_\mathrm{O} \Rightarrow \ell_\mathrm{I}$, we know that $\ell_\mathrm{I} \vee \ell'_\mathrm{O} = \ell_\mathrm{I} \Rightarrow \ell_\mathrm{I} \vee \ell_\mathrm{O}$. Transitivity of $\Rightarrow$ then proves the desired result.

Case $E = \mathsf{return}_\tau\ E'$**:** Inversion on the typing rules proves $\Sigma_\sigma; \cdot; pc; \ell'_\mathrm{I} \vdash s' : \tau \dashv \ell'_\mathrm{O}$ for some ℓ'_I and ℓ'_O where $\ell'_\mathrm{I} \vee \ell'_\mathrm{O} \Rightarrow \ell_\mathrm{O}$. As with the previous case, our inductive hypothesis on E applies, giving us $(\bigwedge \hat{L}) \wedge (\ell'_\mathrm{I} \vee \ell'_\mathrm{O}) \not\Rightarrow \ell_t$. Because $\ell'_\mathrm{I} \vee \ell'_\mathrm{O} \Rightarrow \ell_\mathrm{O}$ and $\ell_\mathrm{O} \Rightarrow \ell_\mathrm{I} \vee \ell_\mathrm{O}$, transitivity of $\Rightarrow$ again gives us the desired result.

Case $E = E'\ \mathsf{with\text{-}lock}\ \ell$**:** Inversion on the typing rules proves $\Sigma_\sigma; \Gamma; pc; \ell'_\mathrm{I} \vdash s' : \tau \dashv \ell'_\mathrm{O}$ where $\ell'_\mathrm{I} \wedge \ell \Rightarrow \ell_\mathrm{I}$ and $\ell'_\mathrm{O} \wedge \ell \Rightarrow \ell_\mathrm{O}$. By the definition of *getLocks*, we know that $\langle s' \mid C\rangle$ must be $\mathcal{T}$-safe with pc and $(\hat{L}, \ell)$. Premise 2 is clearly true of s' since there is no added syntax, so induction on E proves $(\bigwedge(\hat{L}, \ell)) \wedge (\ell'_\mathrm{I} \vee \ell'_\mathrm{O}) \not\Rightarrow \ell_t$.

Using the above facts and the distributive property of the lattice,

$$\begin{aligned}\Big(\bigwedge(\hat{L},\ell)\Big) \wedge (\ell'_{\mathrm{I}} \vee \ell'_{\mathrm{O}}) &= \Big(\bigwedge \hat{L}\Big) \wedge \ell \wedge (\ell'_{\mathrm{I}} \vee \ell'_{\mathrm{O}}) \\ &= \Big(\bigwedge \hat{L}\Big) \wedge ((\ell'_{\mathrm{I}} \wedge \ell) \vee (\ell'_{\mathrm{O}} \wedge \ell)) \\ &\Rightarrow \Big(\bigwedge \hat{L}\Big) \wedge (\ell_{\mathrm{I}} \vee \ell_{\mathrm{O}}).\end{aligned}$$

Transitivity of $\Rightarrow$ finishes the case.

Case $E = E'$ at-pc pc'**:** Here $\langle s' \mid C\rangle$ is $\mathcal{T}$-safe at pc' and $\hat{L}$ and premise 2 clearly holds, so induction on E proves the case.

Case $E =$ ignore-locks-in E'**:** This case is impossible by premise 2.

We now move on to the inductive case on the number of steps. For all cases, Lemma 4.14 ensures that premise 1 remains true after a single step. By inspection on the operational semantics, we can introduce ignore-locks-in terms in only two ways: directly through E-CALL and E-CALLATK and indirectly through E-EVAL. Thus premise 2 inductively holds for all other steps. Similarly, premises 3 and 4 remain true by assumption at top-level. We can therefore directly apply our inductive hypothesis for all steps except E-EVAL, E-CALL, and E-CALLATK. We handle those cases explicitly.

For the case of E-EVAL where $s = \tilde{E}[\tilde{s}]$, we induct on $\tilde{E}$ and the operational semantics.

Case $\tilde{E} = [\cdot]$**:** Here induction on the operational semantic rule proves the case.

Case $\tilde{E} =$ let $x = \tilde{E}'$ in e**:** We now consider two sub-cases: first is the case where $\langle \tilde{E}'[\tilde{s}] \mid C\rangle \longrightarrow^* \langle \tilde{E}''[\text{new } C(\overline{v}).m(\overline{w})] \mid C'\rangle$ and second is the case where no such steps exist. If there is such an evaluation, then all of the inductive hypotheses hold for $\tilde{E}'[\tilde{s}]$, so induction on $\tilde{E}$ prove the case. If there is no such evaluation, inspection on the operational semantics tells us that we can only step s using E-EVAL stepping $\tilde{E}'[\tilde{s}]$ until it steps to a value. Therefore, premise 3, ensures that there is some value v and context C_v such that $\langle \tilde{E}'[\tilde{s}] \mid C\rangle \longrightarrow^{+} \langle v \mid C_v\rangle$. Using E-EVAL on each step gives us

$$\langle s \mid C\rangle \longrightarrow^{+} \langle \mathsf{let}\ x = v\ \mathsf{in}\ e \mid C_v\rangle \longrightarrow^{*} \langle E[\mathsf{new}\ C(\overline{v}).m(\overline{w})] \mid C'\rangle \longrightarrow \langle \tilde{s} \mid C''\rangle.$$

Therefore, $\langle \mathsf{let}\ x = v\ \mathsf{in}\ e \mid C_v\rangle$ satisfies our inductive hypothesis, so induction completes the case.

For the other three possible cases of $\tilde{E}$, the same logic as in the base-case proof above applies.

We now turn to when the step is E-CALL or E-CALLATK. In both cases, the statement $s = \mathsf{new}\ D(\overline{v'}).m'(\overline{w'})$ with $mbody(D, m') = \left(\ell_m, \overline{x}, \overline{\tau'_a}, pc'_1 \gg pc'_2, e, \tau\right)$. If we let $e' = e[\overline{x} \mapsto \overline{w'}, \mathsf{this} \mapsto \mathsf{new}\ D(\overline{v'})]$, this steps to $(\mathsf{return}_{\tau}\ e')\ \mathsf{at\text{-}pc}\ pc'_2$. We handle this in two sub-cases: if $pc'_2 \in \mathcal{T}$ and if $pc'_2 \notin \mathcal{T}$.

If $pc'_2 \in \mathcal{T}$, then METHOD-OK proves that $\ell_m \Rightarrow pc'_2$ and therefore $\ell_m \in \mathcal{T}$. Because $\langle s \mid C\rangle$ is $\mathcal{T}$-safe, the method body e, and hence e', cannot have any subexpressions of the form $\mathsf{ignore\text{-}locks\text{-}in}\ e''$. Therefore the new statement satisfies premise 2 of this lemma, allowing us to apply the inductive hypothesis.

If $pc'_2 \notin \mathcal{T}$, we claim that $(\bigwedge \hat{L}) \wedge pc'_2 \not\Rightarrow \ell_t$, which we prove differently based on the security assumptions of the system: either $\mathcal{A} = \overline{\mathcal{T}}$ is a sublattice, or E-CALLATK is not admissible. In both cases we will apply Lemma 4.16 to the configuration after taking this step. To meet the requirement of the lemma that there is no sub-statement of the form $s'\ \mathsf{at\text{-}pc}\ pc'$, we use $\langle e' \mid (CT, \sigma, (\mathcal{M}, \ell_m), L)\rangle$, noting that this configuration is $\mathcal{T}$-safe with pc'_2 and $\hat{L}$.

Because $pc'_2 \notin \mathcal{T}$, Lemma 4.16 proves that $(\bigwedge \hat{L}) \in \mathcal{A}$. When $\mathcal{A}$ is a sublattice, it is closed under meet, so $(\bigwedge \hat{L}) \wedge pc'_2 \in \mathcal{A}$. By the downward-closed property of $\mathcal{T}$, that means $(\bigwedge \hat{L}) \wedge pc'_2 \not\Rightarrow \ell_t$.

If E-CALLATK is not admissible, Lemma 4.16 proves that $pc'_2 \Rightarrow pc_2 \vee (\bigwedge \hat{L})$. By the definition of meet and the distributive property of the lattice,

$$pc_2' = \left(pc_2 \vee \left(\bigwedge \hat{L}\right)\right) \wedge pc_2' = (pc_2 \wedge pc_2') \vee \left(\left(\bigwedge \hat{L}\right) \wedge pc_2'\right).$$

By assumption on this sub-case, $pc_2' \notin \mathcal{T}$ and therefore $pc_2' \not\Rightarrow \ell_t$, so at least one of the two sides of the join cannot act for ℓ_t. However, the definition of meet gives $pc_2 \wedge pc_2' \Rightarrow pc_2 \Rightarrow \ell_t$. Therefore $(\bigwedge \hat{L}) \wedge pc_2' \not\Rightarrow \ell_t$.

By inversion on the typing rules, if $\Sigma_\sigma; \Gamma; pc; \ell_{\text{I}} \vdash s : \tau \dashv \ell_{\text{O}}$, then $\ell_{\text{O}}' \vee pc_2' \Rightarrow \ell_{\text{O}}$ where ℓ_{O}' is the lock label on $D.m$. In particular, $pc_2' \Rightarrow \ell_{\text{O}} \Rightarrow \ell_{\text{I}} \vee \ell_{\text{O}}$. As a result, $(\bigwedge \hat{L}) \wedge pc_2' \Rightarrow (\bigwedge \hat{L}) \wedge (\ell_{\text{I}} \vee \ell_{\text{O}})$, so transitivity of $\Rightarrow$ proves $(\bigwedge \hat{L}) \wedge (\ell_{\text{I}} \vee \ell_{\text{O}}) \not\Rightarrow \ell_t$. □

Lemma 4.16. *For any statement s, configuration $C = (CT, \sigma, \mathcal{M}, L)$, label pc, and lock list $\hat{L}$, if*

- *$\langle s \mid C \rangle$ is $\mathcal{T}$-safe with pc and $\hat{L}$,*
- *s contains no sub-statements of the form s' at-pc pc',*
- *$\langle s \mid C \rangle \longrightarrow^* \langle E[\textsf{new}\ C(\overline{v}).m(\overline{w})] \mid C' \rangle \longrightarrow \langle s' \mid C'' \rangle$, and*
- *$mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_{\text{O}}} \tau$ with $pc_2 \in \mathcal{T}$,*

then $pc \notin \mathcal{T}$ implies $(\bigwedge \hat{L}) \notin \mathcal{T}$, and $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$ if no step uses E-CallAtk.

Proof. This proof follows by induction on the number of steps. For the base case where $s = E[\textsf{new}\ C(\overline{v}).m(\overline{w})]$, we induct on E to prove $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$.

Case $E = [\cdot]$: Inversion on the typing rules tells us $pc \Rightarrow pc_1$ and inversion on the operational semantics tell us $pc_1 \Rightarrow pc_2 \vee (\bigwedge L)$. By the definition of *getLocks*, $\hat{L} = L$, so transitivity proves the case.

Case $E = E$ at-pc pc': This case is impossible by assumption.

Case $E = E'$ with-lock ℓ: In this case we note that if $s = E[\tilde{s}]$, then $\langle E'[\tilde{s}] \mid C \rangle$ must be $\mathcal{T}$-safe with pc and $(\hat{L}, \ell)$. Therefore, by induction on E, $pc \Rightarrow pc_2 \vee (\bigwedge(\hat{L}, \ell))$. However,

$$\bigwedge(\hat{L}, \ell) = \left(\bigwedge \hat{L}\right) \wedge \ell \Rightarrow \bigwedge \hat{L}.$$

Therefore, by transitivity, $pc \Rightarrow pc_2 \vee (\bigwedge \hat{L})$.

All other cases: The pc is unmodified and $getLocks(\hat{L}, E'[\tilde{s}]) = getLocks(\hat{L}, E[\tilde{s}]) = L$, so a simple inductive application completes the case.

This directly proves the second conclusion in this case. When $pc \notin \mathcal{T}$, the fact that $\mathcal{T}$ is downward-closed means $pc_2 \vee (\bigwedge \hat{L}) \notin \mathcal{T}$. However, $pc_2 \in \mathcal{T}$ and $\mathcal{T}$ is a sublattice, so therefore it must be the case that $(\bigwedge \hat{L}) \notin \mathcal{T}$.

We now move to the inductive step. Lemma 4.14 ensures that $\mathcal{T}$-safety is retained. By inspection on the operational semantic rules, we can introduce new syntax only with E-EVAL, E-IFT, E-IFF, E-CALL, and E-CALLATK. For all other steps, a direct application of the inductive hypothesis proves the lemma. We now prove those cases.

Case E-EVAL**:** This case is by induction on $\tilde{E}$ where $s = \tilde{E}[\tilde{s}]$. If $\tilde{E} = [\cdot]$, induction on the operational semantics completes the case. When $\tilde{E} = \mathsf{let}\ x = \tilde{E}'\ \mathsf{in}\ e$, we must consider whether $\tilde{E}'[\tilde{s}]$ steps to the relevant method call or not. If it does, a direct inductive application proves the case. If it does not, we note that $\langle \tilde{E}'[\tilde{s}] \mid C \rangle \longrightarrow^{+} \langle v \mid C_v \rangle$ for some value v and configuration C_v. This new expression satisfies the premises of our top-level inductive hypothesis, so we can apply that.

By assumption, $\tilde{E} \neq \tilde{E}'\ \mathsf{at\text{-}pc}\ pc'$, and the other possible options are the same as in the base case.

Cases E-IFT **and** E-IFF**:** In this case we note that $s = \mathsf{if}\{pc'\}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Inversion on the typing rules proves that $\Sigma_\sigma; \Gamma; pc'; \ell_I \vdash e_i : \tau \dashv \ell_O$ for both $i = 1, 2$ and $pc \Rightarrow pc'$. Therefore, $\langle e_i \mid C \rangle$ is $\mathcal{T}$-safe with pc' and $\hat{L}$. Moreover, e_1 and e_2 are expressions, so they contain no sub-statements of the form $s''\ \mathsf{at\text{-}pc}\ pc''$, allowing us to apply our inductive hypothesis. If $pc \notin \mathcal{T}$, then because $\mathcal{T}$ is downward-closed, $pc' \notin \mathcal{T}$, so induction proves that $(\bigwedge \hat{L}) \notin \mathcal{T}$.

If E-CallAtk is not admissible, induction proves $pc' \Rightarrow pc_2 \vee (\bigwedge \hat{L})$, so transitivity gets us the desired result.

Case E-Call**:** The premises of E-Call require that $s = \mathsf{new}\ D(\overline{v'}).m'(\overline{w'})$ with $mbody(D, m') = \left(\ell_{m'}, \overline{x}, \overline{\tau'_a}, pc'_1 \gg pc'_2, e, \tau'\right)$ and $pc'_1 \Rightarrow pc'_2 \vee (\bigwedge L)$. Inversion on the typing rules proves that $pc \Rightarrow pc'_1$. Additionally, the statement after the step is $\mathsf{return}_{\tau'}\ (e'\ \mathsf{at\text{-}pc}\ pc'_2)$ for some expression e'.

By Lemma 4.14 the new configuration is $\mathcal{T}$-safe at pc and $\hat{L}$, so inductively, replacing the statement with e' is $\mathcal{T}$-safe at pc'_2 and $\hat{L}$. If $pc'_2 \notin \mathcal{T}$, then by induction $(\bigwedge \hat{L}) \notin \mathcal{T}$. When $pc'_2 \in \mathcal{T}$, the fact that $\mathcal{T}$ is a sublattice means $(\bigwedge \hat{L}) \in \mathcal{T} \implies pc'_2 \vee (\bigwedge \hat{L}) \in \mathcal{T}$. Moreover, the acts-for relations proved in the previous paragraph and transitivity give $pc \Rightarrow pc'_2 \vee (\bigwedge \hat{L})$. $\mathcal{T}$ is also downward-closed, so therefore $(\bigwedge \hat{L}) \in \mathcal{T} \implies pc \in \mathcal{T}$. Hence, if $pc \notin \mathcal{T}$ then $(\bigwedge \hat{L}) \notin \mathcal{T}$, as desired.

If E-CallAtk is never used, induction on the number of steps immediately proves $pc'_2 \Rightarrow pc_2 \vee (\bigwedge \hat{L})$. Combining this result with the flow above gives

$$pc \Rightarrow pc'_2 \vee \left(\bigwedge \hat{L}\right) \Rightarrow \left(pc_2 \vee \left(\bigwedge \hat{L}\right)\right) \vee \left(\bigwedge \hat{L}\right) = pc_2 \vee \left(\bigwedge \hat{L}\right).$$

Case E-CallAtk**:** Inversion on the semantic rules proves $pc'_2 \in \mathcal{A} = \overline{\mathcal{T}}$. Using the same argument as in the E-Call case to apply the inductive hypothesis, induction proves that $(\bigwedge \hat{L}) \notin \mathcal{T}$ regardless of the value of pc. This case is impossible by assumption when E-CallAtk is not taken. □

We formalize the concept of a tail call, which is a call initiated in a tail position of some expression, by defining a *tail context* T which, by construction, does nothing after the call returns.

Definition 4.13 (Tail Context)**.**

$$T \ ::= \ [\cdot] \mid \mathsf{return}_{\tau}\ T \mid T\ \mathsf{with\text{-}lock}\ \ell \mid T\ \mathsf{at\text{-}pc}\ pc$$

The following lemma captures our intuition that a tail context "does nothing".

Lemma 4.17. *If $\langle T[v] \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow \langle s \mid (CT, \sigma', \mathcal{M}', L')\rangle$, then for some tail context T', $s = T'[v]$ and $\sigma = \sigma'$.*

Proof. By simple induction on the operational semantics, noting for E-EVAL that, if $T[v] = E[s']$, then $E = T_1$ and $s' = T_2[v]$ for some tail contexts T_1 and T_2. □

Definition 4.14 (Tail Reentrancy)**.** We say a statement s is in an *ℓ_t-tail-reentrant* state if s is ℓ_t-reentrant—that is, $s = E_0[E_1[E_2[s' \text{ at-pc } pc_3] \text{ at-pc } pc_2] \text{ at-pc } pc_1]$ where $pc_1, pc_3 \Rightarrow \ell_t$ and $pc_2 \not\Rightarrow \ell_t$—and there is some tail context T, evaluation context $\tilde{E}_2$, and label pc_2' such that $pc_2' \not\Rightarrow \ell_t$ and

$$E_1[[\cdot] \text{ at-pc } pc_2] = T[\tilde{E}_2 \text{ at-pc } pc_2']$$

Theorem 4.3. *For any label $\ell_t \in \mathcal{T}$, class table CT, and well-typed heap σ_1, if $\Sigma_{\sigma_1} \vdash CT$ ok complies with locks in $\mathcal{T}$-code, then for any invocation I and heap σ_2 where $\Sigma_{\sigma_1} \vdash I$ and $(I, CT, \sigma_1) \Downarrow \sigma_2$, all ℓ_t-reentrant states in the execution are ℓ_t-tail-reentrant.*

Proof. By Definition 4.5, if $I = (\iota, m(\overline{v}), \ell)$ is an ℓ_t-reentrant invocation in σ_1, there must exists a statement s such that

$$s = E_0\Big[E_1[E_2[s' \text{ at-pc } pc_3] \text{ at-pc } pc_2] \text{ at-pc } pc_1\Big]$$

where $pc_1, pc_3 \Rightarrow \ell_t$ but $pc_2 \not\Rightarrow \ell_t$, and $\langle !\iota.m(\overline{v}) \mid (CT, \sigma_1, \ell, \cdot)\rangle \longrightarrow^* \langle s \mid C\rangle$. We prove by induction on E_1 that s is ℓ_t-tail-reentrant according to Definition 4.14. Specifically, we claim the following.

Claim. *If $s = E_0'[E_1[s'' \text{ at-pc } pc_2]]$ where $innerPc(\ell, E_0') \Rightarrow \ell_t$, then there is some $\tilde{E}_2$, T, and pc_2' such that $E_1[[\cdot] \text{ at-pc } pc_2] = T[\tilde{E}_2 \text{ at-pc } pc_2']$ and $pc_2' \not\Rightarrow \ell_t$.*

Proof of claim. This is a proof by induction on E_1.

Case $E_1 = [\cdot]$: Because $pc_2 \not\Rightarrow \ell_t$ by assumption, letting $pc_2' = pc_2$, $\tilde{E}_2 = [\cdot]$, and $T = [\cdot]$ proves the case.

Case $E_1 = E_1' \ \mathsf{at\text{-}pc}\ pc'$: There are two sub-cases to consider. If $pc' \Rightarrow \ell_t$, then the inductive hypothesis applies by replacing E_0' with $E_0'[[\cdot]\ \mathsf{at\text{-}pc}\ pc']$ and E_1 with E_1'. It then proves that $E_1' = T'[\tilde{E}_2\ \mathsf{at\text{-}pc}\ pc_2']$ for some $pc_2' \not\Rightarrow \ell_t$. Letting $T = T'\ \mathsf{at\text{-}pc}\ pc'$ completes the sub-case.

If $pc' \not\Rightarrow \ell_t$, then letting $\tilde{E}_2 = E_1'$, $pc_2' = pc'$, and $T = [\cdot]$ proves the case.

Case $E_1 = E_1'\ \mathsf{with\text{-}lock}\ \ell$: Replacing E_0' with $E_0'[[\cdot]\ \mathsf{with\text{-}lock}\ \ell]$ and E_1 with E_1', the inductive hypothesis proves $E_1' = T'[\tilde{E}_2\ \mathsf{at\text{-}pc}\ pc_2']$ for some $pc_2' \not\Rightarrow \ell_t$. Letting $T = T'\ \mathsf{with\text{-}lock}\ \ell$ completes the case.

Case $E_1 = \mathsf{return}_\tau\ E_1'$: This case follows from the same logic as the previous case.

Case $E_1 = (\mathsf{let}\ x = E_1'\ \mathsf{in}\ e)$: Let $pc = \mathit{innerPc}(\ell, E_0')$. Lemma 4.14 and induction on the number of steps to get to s prove that, if $\langle !\iota.m(\overline{v}) \mid (CT, \sigma_1, \ell, \cdot)\rangle \longrightarrow^* \langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle$, then it must be the case that each configuration encountered along the way is ℓ_t-safe with ℓ and $\cdot$.

To step to s, there must be some expression e_1 such that

$$\langle !\iota.m(\overline{v}) \mid (CT, \sigma_1, \ell, \cdot)\rangle \longrightarrow^* \langle E_0'[\mathsf{let}\ x = e_1\ \mathsf{in}\ e] \mid (CT, \sigma', \mathcal{M}, L)\rangle$$

and

$$\langle e_1 \mid (CT, \sigma', \mathcal{M}, L)\rangle \longrightarrow^* \langle E[\mathsf{new}\ D(\overline{v'}).m'(\overline{w})] \mid C'\rangle$$

where $\mathit{mtype}(D, m') = \overline{\tau_a} \xrightarrow{\tilde{pc}_1 \gg \tilde{pc}_2; \tilde{\ell}_O} \tilde{\tau}$ such that $\tilde{pc}_1 \not\Rightarrow \ell_t$ and $\tilde{pc}_2 \Rightarrow \ell_t$. Inversion on the typing rules and the safety of $\langle E_0'[\mathsf{let}\ x = e_1\ \mathsf{in}\ e] \mid (CT, \sigma', \mathcal{M}, L)\rangle$ prove that $\Sigma; \Gamma; pc; \ell_I \vdash e_1 : \tau_1 \dashv \ell_O$ for some Σ, Γ, ℓ_I, τ_1, and ℓ_O, where $\ell_O \Rightarrow \ell_I$ and $(\bigwedge \mathit{getLocks}(\cdot, E_0')) \wedge \ell_I \Rightarrow pc \Rightarrow \ell_t$. Moreover, the safety of the configuration guarantees that $\mathit{getLocks}(\cdot, E_0')$ is a prefix of L, so in particular, $\bigwedge L \Rightarrow \bigwedge \mathit{getLocks}(\cdot, E_0')$.

However, Lemma 4.15 proves that, because $\Sigma; \Gamma; pc; \ell_I \vdash e_1 : \tau_1 \dashv \ell_O$, it must be the case that $(\bigwedge L) \wedge (\ell_I \vee \ell_O) \not\Rightarrow \ell_t$. Yet we already proved that $\ell_O \Rightarrow \ell_I$, meaning $\ell_I \vee \ell_O = \ell_I$, and $\bigwedge L \Rightarrow \bigwedge \mathit{getLocks}(\cdot, E'_0)$. Together these prove that $(\bigwedge \mathit{getLocks}(\cdot, E'_0)) \wedge \ell_I \not\Rightarrow \ell_t$. This lack of relationship contradicts the safety result, so this case is impossible.

Case $E_1 = \textsf{ignore-locks-in}\ E'_1$**:** Safety of the configuration, as argued in the previous case, proves that $\ell \vdash_{\mathcal{T}} s\ \textsf{cwl}$. Because, by assumption, $\mathit{innerPc}(\ell, E'_0) \Rightarrow \ell_t$, inversion on the proof rules for $\ell \vdash_{\mathcal{T}} s\ \textsf{cwl}$ demonstrates that this case is impossible. □

Letting $E'_0 = E_0[[\cdot]\ \textsf{at-pc}\ pc_1]$ clearly satisfies the assumptions of the claim. Therefore,

$$s = E_0[T[\tilde{E}_2[s'']\ \textsf{at-pc}\ pc'_2]\ \textsf{at-pc}\ pc_1]$$

for some $pc'_2 \not\Rightarrow \ell_t$. This form satisfies Definition 4.14 and proves the theorem. □

4.12.2 All Tail Reentrancy is Secure

We now present a proof for Theorem 4.4, proving that all tail reentrancy is secure. The proof follows the structure outlined in the proof sketch in Section 4.5.3. It requires one simple lemma and follows essentially as a corollary from a more complicated statement.

Lemma 4.18. *For any type τ and heap-type Σ, there exists a value v such that $\Sigma \vdash v : \tau$.*

Proof. This proof is by induction on the structure of τ. If $\tau = \textsf{unit}^\ell$, $v = ()$. If $\tau = \textsf{bool}^\ell$, $v = \textsf{true}$. If $\tau = (\textsf{ref}\ \tau')^\ell$, $v = \textsf{null}$. If $\tau = C^\ell$, let $\mathit{fields}(C) = \overline{x} : \overline{\tau}$. For each τ_i, by induction, there is some v_i such that $\Sigma \vdash v_i : \tau_i$. Therefore, by NEW, $\Sigma \vdash \textsf{new}\ C(\overline{v}) : C^\ell$. □

For the main proof, we assume the existence of an nat type and constant nat values. This assumption is without loss of generality as natural numbers are simple to encode using objects. The class simply has isZero and previous methods. There are two implementations: zero returns true and this, respectively, while non-zero values have a single field pointing to the previous nat and return false and the value of their one field. We will only use nat to increment and check the value, each of which is simple with this implementation.

Lemma 4.19. *For any class table CT, invocation I, and heaps σ_1 and σ_2, if*

- $\Sigma_{\sigma_1} \vdash CT$ ok *complies with locks in ℓ-code,*
- $\vdash \sigma_1$ wt,
- $\Sigma_{\sigma_1} \vdash I$*, and*
- $(I, CT, \sigma_1) \Downarrow \sigma_2$ *where all ℓ-reentrant states are ℓ-tail-reentrant,*

then there exist CT', $\bar{I}$, σ'_1, and σ'_2 such that

1. $\Sigma_{\sigma'_1} \vdash CT'$ ok *complies with locks in ℓ-code,*
2. $CT \approx_\ell CT'$,
3. $\vdash \sigma'_1$ wt,
4. $\Sigma_{\sigma'_1} \vdash \bar{I}$,
5. $(\bar{I}, CT', \sigma'_1) \Downarrow \sigma'_2$ *are all non-ℓ-reentrant, and*
6. $\sigma_i \approx_\ell \sigma'_i$ *with $\sigma_i \subseteq \sigma'_i$ for both $i = 1, 2$.*

Proof. For notation, let $I = (\ell_I, \iota_I, m_I(\overline{v_I}))$.

Step through the execution of $(I, CT, \sigma_1) \Downarrow \sigma_2$ and create a log of the following relevant events:

1. Calls from low-integrity environments into high-integrity environments.
2. Calls from high-integrity environments into low-integrity environments.
3. Returns from low-integrity environments into high-integrity environments.
4. State modifications from low-integrity environments.

For most events, we will only need to reply the event later, so logging the type of event and the statement that is evaluated is sufficient. For event 2, however, CT' will need to have different code than CT, so there must be a link to the original piece of code. Method calls already have a name and clear location in the code, but if statements can also move from high-integrity to low-integrity and have no names. To support unique tracking, we attach a unique name a to each branch of each conditional in CT. They have the same typing and semantic rules as before, but syntactically include this new annotation, denoted $\mathsf{if}\{pc\}\ v\ \mathsf{then}_{a_1}\ e_1\ \mathsf{else}_{a_2}\ e_2$.

For a semantic step $\langle s \mid (CT, \sigma, \mathcal{M}, L)\rangle \longrightarrow \langle s' \mid (CT, \sigma', \mathcal{M}', L')\rangle$, notationally, let $pc_s = innerPc(\ell_I, s)$ and $pc_{s'} = innerPc(\ell_I, s')$. The following formally defines when each type of event is emitted.

1. When $s = E[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ and $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \tau$, if $pc_s \not\Rightarrow \ell$ and $pc_2 \Rightarrow \ell$, emit $\mathsf{up}(pc_s, \mathsf{new}\ C(\overline{v}).m(\overline{w}), \sigma)$.
2. - When $s = E[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ and $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc_1 \gg pc_2; \ell_O} \tau$, if $pc_s \Rightarrow \ell$ but $pc_2 \not\Rightarrow \ell$, emit $\mathsf{down}(pc_2, C.m)$.
   - When $s = E[\mathsf{if}\{pc\}\ v\ \mathsf{then}_{a_1}\ e_1\ \mathsf{else}_{a_2}\ e_2]$, if $pc_s \Rightarrow \ell$ but $pc \not\Rightarrow \ell$, emit $\mathsf{down}(pc, a_1)$ if $v = \mathsf{true}$ and $\mathsf{down}(pc, a_2)$ if $v = \mathsf{false}$.
3. When $\sigma' = \sigma[\iota \mapsto (v, \tau)] \neq \sigma$, emit $\mathsf{set}(\iota \mapsto (v, \tau))$.
4. When $s = E[v\ \mathsf{at\text{-}pc}\ pc_s]$, if $pc_s \not\Rightarrow \ell$ and $pc_{s'} \Rightarrow \ell$, emit $\mathsf{ret}(v)$.

By inspection on the operational semantics, each step will emit at most one of the above events.

There are several important properties to note about the log. First, the only semantic steps that can change the value of $innerPc(\ell_I, s)$ are E-CALL, E-CALLATK, E-IFT, E-IFF, and E-ATPC. Type preservation (Theorem 4.6) ensures that each statement is well-typed, so if statements can only lower the integrity of the pc, not raise it. Therefore, whenever $pc_s \Rightarrow \ell$ and $pc_{s'} \not\Rightarrow \ell$, the log will contain a down event, and whenever $pc_s \not\Rightarrow \ell$ and $pc_{s'} \Rightarrow \ell$, the log will contain either a up event or ret event. As a result, any two down events must be separated by either an up event or a ret event.

Additionally, the down and ret events must follow a stack discipline as the represent calls and returns. This stack discipline creates a correspondence between each down and exactly one ret, which we will refer to as the "corresponding ret" event.

We now use the log constructed from the execution of $(I, CT, \sigma_1) \Downarrow \sigma_2$ to construct CT', $\bar{I}$, and σ_1'. We will ensure by construction that all conditions hold aside from Condition 6 with σ_2 and σ_2'. We will then argue Condition 6 on σ_2 holds.

Constructing CT', $\bar{I}$, and σ_1'. Initialize $\bar{I} = I$ if $\ell_I \Rightarrow \ell$ and empty otherwise, and initialize $\sigma_1' = \sigma$ and $CT' = CT$. We will add to $\bar{I}$ and σ_1' and modify CT' as the construction progresses.

Step through the log. When a $\mathsf{down}(pc, a)$ event appears, where a can either be $C.m$ or a unique name for the branch of an if statement, note that the log must be of the form $\ldots, \mathsf{down}, \overline{\mathsf{set}}, ev, \ldots$ where ev is either up or ret. If this is the first down event at location a, add a new mapping $\iota_a \mapsto (0, \mathsf{nat}^{pc})$ to σ_1' where ι_a is fresh, meaning $\iota_a \notin \mathrm{dom}(\sigma_1') \cup \mathrm{dom}(\sigma_2)$. Also modify the code at location a in CT'. If this is the first time encountering a, replace the existing code with code that increments ι_a and conditions on it. If this is the nth $\mathsf{down}(pc, a)$ event in the log for $n > 1$, add a new branch to the code in CT' for if $\iota_a \mapsto n$.

The code in the conditional branch for $\iota_a \mapsto n$ will do different things depending on ev. If $ev = \mathsf{ret}(v)$, the code in CT' performs all state modification in $\overline{\mathsf{set}}$ and then returns v. Making these state modification may require constructing new low-integrity methods if pc does not have sufficient integrity for each. Since we know that none of the modified cells are trusted by ℓ, however, making the modifications is always possible using low-integrity code. Moreover, because the state modifications were possible in the original execution without violating locks or entering high-integrity code (there was no up prior to $ev = \mathsf{ret}(v)$), a call graph with the same pc labels where $pc \not\Rightarrow \ell$ for each label must be possible. This guarantees that CT' continues to type-check.

If ev is an up event and this is the nth $\mathsf{down}(pc, a)$ event for location a, then the nth entry into a in CT' simply returns some value v of the appropriate type. By Lemma 4.18, some such well-typed v must exist.

When a $\mathsf{up}(pc, \mathsf{new}\ C(\overline{v}).m(\overline{w}), \sigma)$ event appears in the log, modify both σ'_1 and $\overline{I}$. For σ'_1, add a mapping $\iota \mapsto (\mathsf{new}\ C(\overline{v}), C^{pc})$ for a fresh location $\iota \notin \mathrm{dom}(\sigma'_1) \cup \mathrm{dom}(\sigma_2)$. For $\overline{I}$, add two new invocations. The first performs all state modifications from all set events in the log prior to this up that have not already been performed by a previous invocation. As before, constructing such an invocation may require adding new low-integrity code to CT'. The second invocation added to $\overline{I}$ is $(pc, \iota, m(\overline{w}))$ where ι is the new location added to σ'_1.

Finally, after completing all up and down events in the log, include one final invocation with associated new code to apply any set events not included in any previous invocations.

The construction satisfies all requirements. By construction, the resulting invocations $\overline{I}$ are non-reentrant in CT' with initial state σ'_1. All code changes in CT' were low-integrity and remained well-typed, so $\Sigma_{\sigma'_1} \vdash CT'\ \mathsf{ok}$ complies with locks in ℓ-code and $CT \approx_\ell CT'$. We constructed σ'_1 by adding new well-typed low-integrity mappings to σ_1, meaning $\vdash \sigma'_1\ \mathsf{wt}$, $\sigma_1 \approx_\ell \sigma'_1$, and $\sigma_1 \subseteq \sigma'_1$, as desired. It remains to show that there is a σ'_2 such that $(\overline{I}, CT', \sigma'_1) \Downarrow \sigma'_2$ with $\sigma_2 \approx_\ell \sigma'_2$ and $\sigma_2 \subseteq \sigma'_2$.

Let $\tilde{\sigma}_1, \ldots, \tilde{\sigma}_n$ be the sequence of heaps appearing in the up events in the log. Let $I_1, \ldots, I_n$ be the elements of $\overline{I}$ that call into high-integrity code (note that these are every other element of $\overline{I}$), and let $\tilde{\sigma}'_k$ be the heap provided as input to I_k when executing $(\overline{I}, CT', \sigma'_1) \Downarrow \sigma'_2$. We now argue by induction on k that $\tilde{\sigma}_k \approx_\ell \tilde{\sigma}'_k$ and $\tilde{\sigma}_k \subseteq \tilde{\sigma}'_k$.

For the base case let $k = 1$. There are two sub-cases to consider: if $\ell_I \Rightarrow \ell$ and if it does not. If $\ell_I \Rightarrow \ell$, then $I_1 = I$ and there are no elements of $\overline{I}$ before it, so $\tilde{\sigma}_1 = \sigma_1$ and $\tilde{\sigma}'_1 = \sigma'_1$, meaning the conditions on σ_1 and σ'_1 proved above are precisely the goal. If $\ell_I \not\Rightarrow \ell$, there is one invocation I_0 in $\overline{I}$ before I_1, and it executes only low-integrity code to set mappings. By construction, the code invoked by I_0 performs exactly the modifications to σ'_1 that occurred to σ_1 prior to the up event in the original invocation. Note that some of these modifications may be adding new mappings through using E-REF, which is non-deterministic. Because all mappings in σ'_1 not in σ_1 were taken to be fresh with respect to σ_2 as well, the names used in the original invocation must be free, so we can pick the same names when evaluating to $\tilde{\sigma}'_1$. Therefore, for some set of mappings $\bar{\iota} \mapsto (\bar{v}, \bar{\tau})$, $\tilde{\sigma}_1 = \sigma_1[\bar{\iota} \mapsto (\bar{v}, \bar{\tau})]$ and $\tilde{\sigma}'_1 = \sigma'_1[\bar{\iota} \mapsto (\bar{v}, \bar{\tau})]$. Since $\sigma_1 \approx_\ell \sigma'_1$ and $\sigma_1 \subseteq \sigma'_1$, the same must therefore be true of $\tilde{\sigma}_1$ and $\tilde{\sigma}'_1$, as desired.

Now assume $k > 1$ and, by induction, that $\tilde{\sigma}_{k-1} \approx_\ell \tilde{\sigma}'_{k-1}$ with $\tilde{\sigma}_{k-1} \subseteq \tilde{\sigma}'_{k-1}$. There are two sub-cases to consider depending on whether or not kth up event stems from a ℓ-reentrant call inside the call resulting in the $(k-1)$st up event.

If I_k does *not* correspond to a reentrant call, then I_{k-1} corresponds to a high-integrity call that executed to completion without reentrancy in the original execution. By construction of CT', any part of that execution that operated at low-integrity corresponds to a down in the log, and since none of those produced any high-integrity calls (that would cause reentrancy), they modified the state by incrementing new low-integrity counters and otherwise making the same modifications and returning the same values as the original execution. In particular, the changes to $\tilde{\sigma}'_{k-1}$ needed to achieve the state $\hat{\sigma}'$ after completing I_{k-1}, are updates to new low-integrity counters and the changes to $\tilde{\sigma}_{k-1}$ to achieve the state $\hat{\sigma}$ after completing the original high-integrity call. Because $\tilde{\sigma}_{k-1} \approx_\ell \tilde{\sigma}'_{k-1}$ and $\tilde{\sigma}_{k-1} \subseteq \tilde{\sigma}'_{k-1}$, it must be that $\hat{\sigma} \approx_\ell \hat{\sigma}'$ and $\hat{\sigma} \subseteq \hat{\sigma}'$.

Further, any state modifications made after the high-integrity call returns (and thus after I_{k-1} completes) but before the kth up event (the beginning of I_k) must be made in a low-integrity environment. By the same logic as Lemma 4.9 from the proof of Noninterference, they must be updates to low-integrity state. As a result, each has a corresponding set event in the log, denoted $\mathsf{set}(\iota \mapsto (\overline{v}, \overline{\tau}))$. The extra low-integrity invocation added to $\overline{I}$ before I_k makes exactly these modifications to the state. Therefore, $\tilde{\sigma}_k = \hat{\sigma}[\iota \mapsto (\overline{v}, \overline{\tau})]$ and $\tilde{\sigma}'_k = \hat{\sigma}'[\iota \mapsto (\overline{v}, \overline{\tau})]$. The desired result follows from the above-proved correspondence of $\hat{\sigma}$ and $\hat{\sigma}'$.

Lastly, consider the case where I_k corresponds to a reentrant call inside the call that I_{k-1} corresponds to. That is, the has the form $\ldots, \mathsf{up}_{k-1}, \overline{ev}, \mathsf{down}(pc, a), \overline{\mathsf{set}}, \mathsf{up}_k, \ldots$ where $\overline{ev}$ contains no up events. In this case, the code in CT' created to replace the $\mathsf{down}(pc, a)$ event simply returns an arbitrary value of the correct type without modifying the state. Because we assumed all reentrancy was tail-reentrancy, this means up_k occurred when stepping a term of the form

$$E_0[T[E_2[\mathsf{new}\ C(\overline{v}).m(\overline{w})]\ \mathsf{at\text{-}pc}\ pc_2]\ \mathsf{at\text{-}pc}\ pc_1]$$

where $pc_1 \Rightarrow \ell$, $pc_2 \not\Rightarrow \ell$, and $mtype(C, m) = \overline{\tau_a} \xrightarrow{pc'_2 \gg pc_3; \ell_O} \tau$ with $pc_3 \Rightarrow \ell$.

In CT', we replaced the code corresponding to $E_2[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ with code that returns an arbitrary value of the correct type, and splitting the invocations means inside I_{k-1}, E_0 will be empty. Therefore, by Lemma 4.17, once $E_2[\mathsf{new}\ C(\overline{v}).m(\overline{w})]$ evaluates to some value v, $T[v]$ will evaluate to v with no changes to the state. Similarly, I_{k-1} will return the arbitrary value returned in CT' without examining it or modifying the state at all. That means that the change from $\tilde{\sigma}'_{k-1}$ to $\hat{\sigma}'$, the heap when I_{k-1} returns, is, as before, updates to new low-integrity counters coupled with exactly the change from $\tilde{\sigma}_{k-1}$ to the heap $\hat{\sigma}$ when the $\mathsf{down}(pc, a)$ event occurred. The low-integrity state modifications in the extra invocation before I_k are again those made by the low-integrity code in CT before the call corresponding to $\mathsf{up}(pc', \mathsf{new}\ C(\overline{v}).m(\overline{w}), \tilde{\sigma}_k)$. By the same argument as before, $\tilde{\sigma}_k \approx_\ell \tilde{\sigma}'_k$ and $\tilde{\sigma}_k \subseteq \tilde{\sigma}'_k$, as desired.

We have now shown that the state before each I_k is a ℓ-equivalent superset of the state before the corresponding call in the original execution. To see that this result extends to σ_2 and σ'_2, note that the logic above for non-reentrant calls applies to show that the state after completing I_n is a ℓ-equivalent superset of the state after completing the call that generated the final up event in the original execution. There may be further low-integrity code in the original execution that modifies the state, but all such modifications generate set events and are updated by the final invocation in $\overline{I}$ as described above. Therefore, again, σ_2 and σ'_2 are acquired by making identical modifications to the heap after the return of the final high-integrity call, thereby proving $\sigma_2 \approx_\ell \sigma'_2$ and $\sigma_2 \subseteq \sigma'_2$. □

Theorem 4.4. *Let CT be a class table, σ_1 and σ_2 be well-typed heaps, and I be an invocation such that $(I, CT, \sigma_1) \Downarrow \sigma_2$ where all ℓ-reentrant states are ℓ-tail-reentrant. For any ℓ-integrity predicates P and Q, if $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$ and $P(\sigma_1)$, then $Q(\sigma_2)$.*

Proof. Lemma 4.19 proves that there exists CT', $\overline{I}$, σ'_1, and σ'_2 with the properties stated in the lemma. Because P is a ℓ-integrity predicate and $\sigma_1 \approx_\ell \sigma'_1$, the assumption that $P(\sigma_1)$ means $P(\sigma'_1)$. The definition of $\Sigma_{\sigma_1} \vDash^1_\ell \{P\}\ CT\ \{Q\}$, coupled with $CT \approx_\ell CT'$ and $\Sigma_{\sigma_1} \subseteq \Sigma_{\sigma'_1}$ mean that since $P(\sigma'_1)$ holds, $Q(\sigma'_2)$ must hold. Finally, since $\sigma_2 \approx_\ell \sigma'_2$, the fact that Q is also a ℓ-integrity predicate proves $Q(\sigma_2)$. □

CHAPTER 5

CONCLUSION

This dissertation has explored three different ways for decentralized systems to maintain integrity in the presence of distrusting parties. Each strategy addresses a different type of vulnerability.

We began in Chapter 2 with Solidus, a protocol for anonymous transactions in a bank-intermediated system. Solidus relies partially on the increasingly popular idea of replicating data to maintain system integrity across transactions. By placing only encryptions of account balances on its public ledger, however, Solidus creates an information asymmetry and requires careful use of cryptographic tools to establish the integrity of each individual transaction without compromising confidentiality. One of those tools is the Publicly Verifiable Oblivious RAM Machine (PVORM), The PVORM is a new cryptographic primitive that supports storing and updating data such that only authorized parties can see the data or access patterns, but anyone can verify that data updates conform to a given specification. Giving each bank a separate PVORM allows Solidus to separate banks effectively and allow for efficient transaction processing.

Solidus demonstrates the power of combining cryptographic building blocks, but it also illustrates the precision, care, and expert reasoning required to design a protocol and prove its security. Defining and using a PVORM helps to make the construction more modular, but much of the security still relies on analysis that could break in unexpected ways with small protocol modifications.

Chapters 3 and 4 provided more principled ways to reason about program security. The IFC techniques they developed provide a structure for compositional reasoning about integrity in real-world systems that include endorsement of both data and control flow. Nonmalleable Information Flow Control (Chapter 3) deepened the existing connection between confidentiality and integrity. It identified and formalized a class of potential attacks enabled by endorsing secret data, and it presented a simple condition to provably eliminate such vulnerabilities. Chapter 4 recast the reentrancy attacks that have cost blockchain smart contract systems tens of millions of dollars [127, 131] as integrity failures. The vulnerabilities result from an attacker violating the implicit assumption of many services that requests will follow a call-and-return pattern, where one completes before a second begins. Services that endorse control flow in an unconstrained manner cannot safely make this assumption, drastically complicating reasoning about correctness and security. Chapter 4 formalized this formerly implicit assumption and a security condition stating that it is not violated. It then provided a technique for making the assumption explicit and provably enforcing it with a combination of static and dynamic locks on integrity levels.

Together these three works provide tools and insights for enforcing integrity in decentralized systems with mutual distrust. Notably, Chapter 2 uses very different tools and techniques from Chapters 3 and 4, and both approaches have notable shortcomings. The applied cryptographic techniques of Solidus are difficult to use and their security is fragile in the face of protocol changes. IFC techniques provide robust compositional guarantees, but operate at a very high level, often abstracting away details like how to prevent an attacker from directly viewing data marked as secret or writing to memory marked as trusted.

The strengths of the two approaches are complementary. Combining them shows promise for building executable protocols with compositional security guarantees that are easier to understand. Systems like Wysteria [135] and λ-Symphony [56] have used this approach for confidentiality, and Viaduct [4] has made strides toward incorporating integrity as well.

However, there is still much work to be done on this connection. Permissive but meaningful information security conditions like NMIF assume deterministic single-threaded executions and need to be extended and modified to accommodate the probabilistic guarantees of underlying cryptographic protocols. It is my hope that the work in this dissertation will help provide foundations and direction for building these connections, making both IFC techniques and powerful cryptographic tools easier to deploy in real-world systems.

BIBLIOGRAPHY

[1] Martín Abadi. Access control in a core calculus of dependency. In *11th International Conference on Functional Programming (ICFP '06)*, pages 263–273, September 2006. doi: 10.1145/1159803.1159839.

[2] Martín Abadi. Variations in access control logic. In *2nd International Conference on Deontic Logic in Computer Science*, pages 96–109, 2008. doi: 10.1007/978-3-540-70525-3_9.

[3] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, January 1999. doi: 10.1145/292540.292555.

[4] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. Viaduct: An extensible, optimizing compiler for secure distributed programs. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '21)*, June 2021. doi: 10.1145/3453483.3454074.

[5] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Taming callbacks for smart contract modularity. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), November 2020. doi: 10.1145/3428277.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd Hardware and Architectural Support for Security and Privacy (HASP '13)*, June 2013.

[7] Apache Software Foundation. Apache ZooKeeper (Version 3.4.9). `https://zookeeper.apache.org/`, 2016.

[8] Owen Arden. *Flow-Limited Authorization*. PhD thesis, Cornell University, 2016.

[9] Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *29th IEEE Computer Security Foundations Symposium (CSF '16)*, June 2016. doi: 10.1109/CSF.2016.17.

[10] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *33rd IEEE Symposium on Security and Privacy (Oakland '12)*, May 2012. doi: 10.1109/SP.2012.22.

[11] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *28th IEEE Computer Security Foundations Symposium (CSF '15)*, July 2015. doi: 10.1109/CSF.2015.42.

[12] Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science (LMCS)*, 7(3), September 2011. doi: 10.2168/LMCS-7(3:17)2011.

[13] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th IEEE Symposium on Foundations of Computer Science (FOCS '04)*, October 2004. doi: 10.1109/FOCS.2004.71.

[14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *33rd International Cryptology Conference (CRYPTO '13)*, August 2013. doi: 10.1007/978-3-642-40084-1_6.

[15] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *35th IEEE Symposium on Security and Privacy (Oakland '14)*, May 2014. doi: 10.1109/SP.2014.36.

[16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security '14)*, August 2014. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson`.

[17] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *36th IEEE Symposium on Security and Privacy (Oakland '15)*, May 2015. doi: 10.1109/SP.2015.25.

[18] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *11th Workshop on Programming Languages and Analysis for Security (PLAS '16)*, October 2016. doi: 10.1145/2993600.2993611.

[19] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp, Bedford, MA, 1977. URL `https://apps.dtic.mil/sti/pdfs/ADA039324.pdf`.

[20] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable

resources. `https://developers.diem.com/docs/technical-papers/move-paper/`, May 2020. Accessed March 2021.

[21] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only Oblivious RAM. In *21st ACM Conference on Computer and Communication Security (CCS '14)*, November 2014. doi: 10.1145/2660267.2660313.

[22] Tamás Blummer. Personal communication with Tamás Blummer, Chief Ledger Architect, Digital Asset Holdings, 2016.

[23] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In *19th International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt '00)*, May 2000. doi: 10.1007/3-540-45539-6_31.

[24] BouncyCastle 1.55. Bouncy Castle Crypto APIs (Version 1.55). `https://www.bouncycastle.org/`, 2016.

[25] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. `https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`, 22 July 2017. Accessed March 2021.

[26] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, June 2020. doi: 10.1145/3385412.3385990.

[27] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In *37th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '10)*, January 2010. doi: 10.1145/1706299.1706349.

[28] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *20th International Conference on Functional Programming (ICFP '15)*, August 2015. doi: 10.1145/2784731.2784758.

[29] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In *17th International Cryptology Conference (CRYPTO '97)*, August 1997. doi: 10.1007/BFb0052252.

[30] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *24th International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt '05)*, May 2005. doi: 10.1007/11426639_18.

[31] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *28th IEEE Symposium on Security and Privacy (Oakland '07)*, May 2007. doi: 10.1109/SP.2007.15.

[32] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized schnorr proofs. In *28th International Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt '09)*, April 2009. doi: 10.1007/978-3-642-01001-9_25.

[33] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. iUC: Flexible universal composability made simple. In *25th International Conference on The Theory and Application of Cryptology and Information Security (AsiaCrypt '19)*, December 2019. doi: 10.1007/978-3-030-34618-8_7.

[34] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *42nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*, October 2001. doi: 10.1109/SFCS.2001.959888.

[35] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, February 1999. URL `https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance`.

[36] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *24th ACM Conference on Computer and Communication Security (CCS '17)*, October 2017. doi: 10.1145/3133956.3134054.

[37] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via PVORM. In *24th ACM Conference on Computer and Communication Security (CCS '17)*, October 2017. doi: 10.1145/3133956.3134010.

[38] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. Securing smart contracts with information flow. In *3rd International Symposium on Foundations and Applications of Blockchain (FAB '20)*, May 2020.

[39] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. Compositional security for reentrant applications. In *42nd IEEE Symposium on Security and Privacy (Oakland '21)*, May 2021. doi: 10.1109/SP40001.2021.00084.

[40] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. Technical Report arXiv:1802.07242, February 2018. URL `https://arxiv.org/abs/1802.07242`.

[41] David Chaum. Blind signatures for untraceable payments. In *3rd International Cryptology Conference (CRYPTO '83)*, August 1983. doi: 10.1007/978-1-4757-0602-4_18.

[42] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *8th International Cryptology Conference (CRYPTO '88)*, August 1988. doi: 10.1007/0-387-34799-2_25.

[43] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW '06)*, July 2006. doi: 10.1109/CSFW.2006.11.

[44] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *21st IEEE Computer Security Foundations Symposium (CSF '08)*, June 2008. doi: 10.1109/CSF.2008.12.

[45] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, June 2009. doi: 10.1145/1542476.1542483.

[46] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security (JCS)*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393.

[47] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3), November 2020. doi: 10.1145/3417516.

[48] ConsenSys Diligence. Uniswap audit. `https://github.com/ConsenSys/Uniswap-audit-report-2018-12#31-liquidity-pool-can-be-stolen-in-some-tokens-eg-erc-777-29`, January 2019. Accessed March 2021.

[49] CVE-2014-1772. CVE-2014-1772. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1772`, 29 January 2014. Accessed March 2021.

[50] CVE-2018-8174. CVE-2018-8174. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8174`, 14 March 2018. Accessed March 2021.

[51] CWE-1265. CWE-1265: Unintended reentrant invocation of non-reentrant code via nested calls. `https://cwe.mitre.org/data/definitions/1265.html`, 20 December 2018. Accessed March 2021.

[52] Phil Daian. Analysis of the DAO exploit. `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, 18 June 2016. Accessed March 2021.

[53] Ivan Damgård. On Σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 2002.

[54] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *2016 Network and Distributed System Security Symposium (NDSS '16)*, February 2016.

[55] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: Building Zerocoin from a succinct pairing-based proof system. In *1st ACM Workshop on Language Support for Privacy-Enhancing Technologies*, 2013. doi: 10.1145/2517872.2517878.

[56] David Darais, David Heath, Ryan Estes, William Harris, , and Michael Hicks. λ-Symphony: A concise language model for MPC. `https://www.cs.umd.edu/~mwh/papers/mwh.html`, July 2020. Accessed June 2021.

[57] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE*

Computer Security Foundations Symposium (CSF '21), June 2021. doi: 10.1109/CSF51468.2021.00004.

[58] Michael del Castillo. Overstock just closed its first day of blockchain stock trading. *Coindesk*, 16 December 2016. URL `https://www.coindesk.com/overstock-first-day-blockchain-stock-trading`.

[59] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976. doi: 10.1145/360051.360056.

[60] Diem Association. Diem white paper. `https://www.diem.com/en-us/white-paper/`, April 2020. Accesses May 2021.

[61] Digital Asset. Digital asset plaform. `www.digitalasset.com`, 2017.

[62] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security '04)*, August 2004. URL `https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router`.

[63] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Review*, 45(4):727–784, 2003. doi: 10.1137/S0036144503429856.

[64] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[65] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *20th ACM*

SIGOPS Symposium on Operating Systems Principles (SOSP '05), October 2005. doi: 10.1145/1095810.1095813.

[66] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *21st ACM Conference on Computer and Communication Security (CCS '14)*, November 2014. doi: 10.1145/2660267.2660343.

[67] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *6th International Cryptology Conference (CRYPTO '86)*, August 1986. doi: 10.1007/3-540-47721-7_12.

[68] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security '20)*, August 2020. URL `https://www.usenix.org/conference/usenixsecurity20/presentation/frank`.

[69] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012. URL `https://www.usenix.org/system/files/conference/osdi12/osdi12-final-35.pdf`.

[70] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *26th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '17)*, 2017. doi: 10.1145/3132747.3132757.

[71] Glasgow Haskell Compiler. The Glasgow Haskell Compiler. URL `https://www.haskell.org/ghc/`.

[72] Joseph A. Goguen and José Meseguer. Security policies and security models. In *3rd IEEE Symposium on Security and Privacy (Oakland '82)*, April 1982. doi: 10.1109/SP.1982.10014.

[73] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *7th Principles of Security and Trust (POST '18)*, April 2018. doi: 10.1007/978-3-319-89722-6_10.

[74] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of Ethereum smart contracts. In *30th International Conference on Computer Aided Verification (CAV '18)*, July 2018. doi: 10.1007/978-3-319-96145-3_4.

[75] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, December 2017. doi: 10.1145/3158136.

[76] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium (CSF '12)*, June 2012. doi: 10.1109/CSF.2012.19.

[77] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted Bitcoin-compatible anonymous

payment hub. In *2017 Network and Distributed System Security Symposium (NDSS '17)*, February 2017.

[78] Gesine Hinterwälder, Christian T. Zenger, Foteini Baldimtsi, Anna Lysyanskaya, Christof Paar, and Wayne P. Burleson. Efficient e-cash in practice: NFC-based payments for public transportation systems. In *13th Privacy Enhancing Technologies Symposium (PETS '13)*, July 2013. doi: 10.1007/978-3-642-39077-7_3.

[79] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972. doi: 10.1007/BF00289507.

[80] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *2nd Hardware and Architectural Support for Security and Privacy (HASP '13)*, June 2013.

[81] Scott Hudson, Frank Flannery, C. Scott Ananian, and Michael Petter. CUP 0.11b: Construction of Useful Parsers. Software release, June 2014. URL `http://www2.cs.tum.edu/projects/cup`.

[82] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC '10)*, June 2010. URL `https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems`.

[83] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming*

Languages and Systems (TOPLAS), 23(3):396–450, May 2001. doi: 10.1145/503502.503505.

[84] Intel SGX. *Intel® Software Guard Extensions SDK*. Intel Corporation, 2016. Accessed February 2017.

[85] Markus Jakobsson and Ari Juels. Millimix: Mixing in small batches. Technical report, DIMACS Technical report 99-33, 1999.

[86] Tom Elvis Jedusor. MimbleWimble. `https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt`, 19 July 2016. Accessed March 2021.

[87] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: A programming language for authorization and audit. In *13th International Conference on Functional Programming (ICFP '08)*, September 2008. doi: 10.1145/1411204.1411212.

[88] Shaul Kfir. Personal communication with Shaul Kfir, CTO, Digital Asset Holdings, 2016.

[89] Gerwin Klein, Steve Rowe, and Regis Decamp. JFlex 1.8.2. Software release, `https://jflex.de`, May 2020. URL `https://jflex.de`.

[90] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *37th IEEE Symposium on Security and Privacy (Oakland '16)*, May 2016. doi: 10.1109/SP.2016.55.

[91] krdlab. Haskell servant example. `https://github.com/krdlab/examples`, December 2014.

[92] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, October 2007. doi: 10.1145/1294261.1294293.

[93] Johannes Krupp and Christian Rossow. teEther: Gnawing at Ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security '18)*, August 2018. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/krupp`.

[94] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998. doi: 10.1145/279227.279229.

[95] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, June 2020. doi: 10.1145/3385412.3385982.

[96] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *32nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '05)*, January 2005. doi: 10.1145/1040305.1040319.

[97] Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW '06)*, July 2006. doi: 10.1109/CSFW.2006.13.

[98] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security (JCS)*, 25(4–5):319–321, May 2017. doi: 10.3233/JCS-0559.

[99] Denis Lukianov. Compact confidential transactions. `http://voxelsoft.com/dev/cct.pdf`, 2015.

[100] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *23rd ACM Conference on Computer and Communication Security (CCS '16)*, October 2016. doi: 10.1145/2976749.2978309.

[101] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *36th IEEE Symposium on Security and Privacy (Oakland '15)*, May 2015. doi: 10.1109/SP.2015.28.

[102] Tom Magrino, Jed Liu, Owen Arden, Chinawat Isradisaikul, and Andrew C. Myers. Jif 3.5: Java information flow. Software release, June 2016. URL `https://www.cs.cornell.edu/jif`.

[103] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *2nd Asian Symposium on Programming Languages and Systems (APLAS '04)*, November 2004. doi: 10.1007/978-3-540-30477-7_9.

[104] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. `https://bitcointalk.org/?topic=279249`, August 2013.

[105] Gregory Maxwell. Confidential transactions. `https://people.xiph.org/~greg/confidential_values.txt`, 2013.

[106] Gregory Maxwell and Andrew Poelstra. Borromean ring signatures. `https://github.com/Blockstream/borromean_paper`, 2015.

[107] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions

and software model for isolated execution. In 2nd *Hardware and Architectural Support for Security and Privacy (HASP '13)*, June 2013.

[108] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of bitcoins: Characterizing payments among men with no names. In *13th Internet Measurement Conference (ICM '13)*, October 2013. doi: 10.1145/2504730.2504747.

[109] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *31st IEEE Symposium on Security and Privacy (Oakland '10)*, May 2010. doi: 10.1109/SP.2010.36.

[110] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *34th IEEE Symposium on Security and Privacy (Oakland '13)*, 2013. doi: 10.1109/SP.2013.34.

[111] Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, June 2018. doi: 10.1145/3192366.3192375.

[112] Monero. Monero. `www.getmonero.org`, Referenced May 2017.

[113] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *26th IEEE Computer Security Foundations Symposium (CSF '13)*, June 2013. doi: 10.1109/CSF.2013.8.

[114] Malte Möser, Rainer Böhme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *2013 APWG eCrime researchers summit (eCrime '13)*, September 2013. doi: 10.1109/eCRS.2013.6805780.

[115] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, , and Nicolas Christin. An empirical analysis of traceability in the Monero blockchain. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2018(3):143–163, 2018.

[116] Daniel Mulligan. Know your customer regulations and the international banking system: towards a general self-regulatory regime. *Fordham Int'l LJ*, 22:2324, 1998.

[117] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, January 1999. doi: 10.1145/292540.292561.

[118] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Security and Privacy (Oakland '98)*, May 1998. doi: 10.1109/SECPRI.1998.674834.

[119] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, October 2000. doi: 10.1145/363516.363526.

[120] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security (JCS)*, 14(2):157–196, 2006. doi: 10.3233/JCS-2006-14203.

[121] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2009.

[122] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *32nd*

IEEE Symposium on Security and Privacy (Oakland '11), May 2011. doi: 10.1109/SP.2011.12.

[123] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *34th Annual Computer Security Applications Conference*, December 2018. doi: 10.1145/3274694.3274743.

[124] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, June 2014. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[125] Oracle Corporation. Java SE version 15 API specification. java.util.Map#computeIfAbsent. `https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Map.html#computeIfAbsent(K, java.util.function.Function)`, September 2020. Accessed March 2021.

[126] Parity Technologies. A postmortem on the parity multi-sig library self-destruct. `https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`, 15 November 2017. Accessed March 2021.

[127] PeckShield. Uniswap/Lendf.Me hacks: Root cause and loss analysis. `https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09`, April 2020. Accessed March 2021.

[128] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *11th International Cryptology Conference (CRYPTO '91)*, August 1991. doi: 10.1007/3-540-46766-1_9.

[129] Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.

[130] Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *16th IEEE Symposium on Security and Privacy (Oakland '95)*, May 1995. doi: 10.1109/SECPRI.1995.398926.

[131] Nathaniel Popper. A hacking of more than $50 million dashes hopes in the world of virtual currency. *The New York Times*, 17 June 2016.

[132] François Pottier and Sylvain Conchon. Information flow inference for free. In *5th International Conference on Functional Programming (ICFP '00)*, September 2000. doi: 10.1145/351240.351245.

[133] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1): 117–158, January 2003. doi: 10.1145/596980.596983.

[134] Protocol Labs. Filecoin: A decentralized storage network. `https://filecoin.io/filecoin.pdf`, July 19, 2017. Accessed March 2021.

[135] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *35th IEEE Symposium on Security and Privacy (Oakland '14)*, May 2014. doi: 10.1109/SP.2014.48.

[136] Ripple. Ripple. `https://ripple.com/`, 2017.

[137] Daniel S. Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only ORAM. In *24th ACM Conference on Computer and Communication Security (CCS '17)*, October 2017. doi: 10.1145/3133956.3134051.

[138] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans-*

actions on Information and System Security (TISSEC), 15(1), March 2012. doi: 10.1145/2133375.2133377.

[139] Andrew W. Roscoe and Michael H. Goldsmith. What is intransitive noninterference? In *12th IEEE Computer Security Foundations Workshop (CSFW '99)*, June 1999. doi: 10.1109/CSFW.1999.779776.

[140] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, June 2009. doi: 10.1145/1542476.1542484.

[141] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In *19th European Symposium on Research in Computer Security (ESORICS '14)*, 2014. doi: 10.1007/978-3-319-11212-1_20.

[142] John Rushby. Noninterference, transitivity and channel-control security policies. Technical Report CSL-92-02, SRI, December 1992. URL `http://www.csl.sri.com/papers/csl-92-2/`.

[143] Rust 2020. The Rust standard library, version 1.48.0. Enum std::collections::hash_map::Entry.or_insert_with. `https://doc.rust-lang.org/std/collections/hash_map/enum.Entry.html#method.or_insert_with`, November 2020. Accessed March 2021.

[144] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. doi: 10.1109/JSAC.2002.806121.

[145] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *International Symposium on Software Security*, November 2003. doi: 10.1007/978-3-540-37621-7_9.

[146] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW '05)*, June 2005. doi: 10.1109/CSFW.2005.15.

[147] Ravi S. Sandhu. Role-based access control. In *Advances in Computers*, volume 46, pages 237–286. Elsevier, 1998. doi: 10.1016/S0065-2458(08)60206-5.

[148] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991. doi: 10.1007/BF00196725.

[149] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 218–219, 2018.

[150] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, October 2019. doi: 10.1145/3360611.

[151] Servant Contributors. Servant – a type-level web DSL. `http://haskell-servant.readthedocs.io/`, 2016.

[152] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007. doi: 10.1145/1315245.1315313.

[153] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *17th International Conference on The Theory and Application of Cryptology and Information Security (AsiaCrypt '11)*, September 2011. doi: 10.1007/978-3-642-25385-0_11.

[154] Jeremy Siek and Walid Taha. Gradual typing for objects. In *21st European Conference on Object-Oriented Programming (ECOOP '07)*, July 2007. doi: 10.1007/978-3-540-73589-2_2.

[155] Solidity. Solidity documentation. Release 0.7.5. `https://docs.soliditylang.org/en/v0.7.5/`, November 18 2020. Accessed December 2020.

[156] Solidity. Solidity security considerations. `https://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern`, 2021. Accessed March 2021.

[157] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, June 2016. doi: 10.1145/2908080.2908092.

[158] Jon Southurst. Blockchain's SharedCoin users can be identified, says security expert. *CoinDesk*, 10 June 2014. URL `https://www.coindesk.com/blockchains-sharedcoin-users-can-identified-says-security-expert`.

[159] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *4th ACM SIGPLAN Haskell Symposium (HASKELL '11)*, September 2011. doi: 10.1145/2034675.2034688.

[160] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *20th ACM Conference on Computer and Communication Security (CCS '13)*, November 2013. doi: 10.1145/2508859.2516660.

[161] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *19th IEEE Computer Security Foundations Workshop (CSFW '06)*, July 2006. doi: 10.1109/CSFW.2006.17.

[162] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. Technical Report arXiv:1906.08936, August 2020. URL `https://arxiv.org/abs/1906.08936`.

[163] The Open Group. SOA standards. `https://publications.opengroup.org/standards/soa`, 2020. Accessed December 2020.

[164] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P '17)*, 2017. doi: 10.1109/EuroSP.2017.28.

[165] Matthew Trudeau. Personal communication with Matthew Trudeau, President, TradeWind Markets, 2016.

[166] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, accountable mixes for Bitcoin. In *19th Financial Cryptography and Data Security (FC '15)*, 2015. doi: 10.1007/978-3-662-48051-9_9.

[167] Ron van der Meyden. What, indeed, is intransitive noninterference? In *12th*

European Symposium on Research in Computer Security (ESORICS '07), September 2007. doi: 10.1007/978-3-540-74835-9_16.

[168] Paul Walker and Phil J. Venables. Cryptographic currency for securities settlement. U.S. Patent Application 20150332395, November 2015.

[169] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich–Ostrovsky lower bound. In *22nd ACM Conference on Computer and Communication Security (CCS '15)*, October 2015. doi: 10.1145/2810103.2813634.

[170] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. It's my privilege: Controlling downgrading in DC-labels. In *11th International Workshop on Security and Trust Management (STM '15)*, September 2015.

[171] Douglas Wikström. Simplified universal composability framework. In *13th IACR Theory of Cryptography Conference (TCC '16)*, January 2016. doi: 10.1007/978-3-662-49096-9_24.

[172] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *11th IEEE Symposium on Security and Privacy (Oakland '90)*, May 1990. doi: 10.1109/RISP.1990.63846.

[173] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[174] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi: 10.1006/inco.1994.1093.

[175] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *39th ACM SIGPLAN Sympo-*

sium on Principles of Programming Languages (POPL '12), January 2012. doi: 10.1145/2103656.2103669.

[176] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, June 2019. doi: 10.1109/CSF.2019.00026.

[177] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, June 2001. doi: 10.1109/CSFW.2001.930133.

[178] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, August 2002. doi: 10.1145/566340.566343.

[179] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, April 2008. URL `https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/zeldovich/zeldovich.pdf`.

[180] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011. doi: 10.1145/2018396.2018419.

[181] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. SHErrLoc: A static holistic error locator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(4), August 2017. doi: 10.1145/3121137.

[182] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town Crier: An authenticated data feed for smart contracts. In *23rd ACM Conference on Computer and Communication Security (CCS '16)*, October 2016. doi: 10.1145/2976749.2978326.

[183] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW '05)*, June 2005. doi: 10.1109/CSFW.2005.16.

[184] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3): 67–84, March 2007. doi: 10.1007/s10207-007-0019-9.

[185] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *24th IEEE Symposium on Security and Privacy (Oakland '04)*, May 2003. doi: 10.1109/SECPRI.2003.1199340.