

PROTOCOLS FOR CONNECTING BLOCKCHAINS WITH OFF-CHAIN 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

Fan Zhang

August 2020

© 2020 Fan Zhang
ALL RIGHTS RESERVED

PROTOCOLS FOR CONNECTING BLOCKCHAINS WITH OFF-CHAIN SYSTEMS

Fan Zhang, Ph.D.

Cornell University 2020

Smart contracts are programs that execute on blockchains. Their strong security properties (e.g., transparency, tamper-resistance, and censorship-resistance) have attracted significant attention and investment (\$31B via ICOs as of 2019), but most of their real-world uses, such as tokens, exercise little of smart contracts' potential power. A key reason is a disconnection from the real world: There is currently no secure, decentralized way to faithfully convey real-world states to blockchains. Worse yet, smart contracts inherit blockchains' lack of confidentiality and poor efficiency.

This thesis introduces solutions to these problems by **connecting blockchains with off-chain systems**. The systems presented in this thesis advance the state of the art of smart contract capabilities. Specifically, this thesis explores three research directions: (1) **authenticated data oracles** that enable faithful representation of real-world states on blockchains. (2) **decentralized secret storage** that endows smart contracts with privacy by storing secrets and performing computation in off-chain committees. (3) **resource-efficient consensus** that achieves Proof of Work style consensus while avoiding wasteful computation. Beyond the scholarly contributions, several works in this thesis have seen industry adoption.

BIOGRAPHICAL SKETCH

Fan Zhang received his B.S. in Electronic Engineering from Tsinghua University in 2014. Since then, he has pursued a Ph.D. in Computer Science at Cornell University under the supervision of Prof. Ari Juels. He is a member of IC3 (Initiative of Cryptocurrencies and Contract) and an IBM Ph.D. Fellow for 2018–2020. His research interest is in the security of decentralized systems, in particular those enabled by blockchains and trusted execution environments (TEEs). In this line of work, Fan has made various types of contribution, including pioneering the study of fusing TEEs and blockchains, developing novel decentralized cryptographic protocols with orders-of-magnitude better performance, and impacting broader community by releasing open source tools and setting up public services. Several of his works have been adopted by industry.

This thesis is dedicated to my family, including Pixie “Pipi” Lian, the cat.

ACKNOWLEDGEMENTS

When applying to grad school many years back, I had a tough time calculating whether pursuing a Ph.D. degree would be optimal. In retrospect, my calculation was mostly naïve (e.g., I figured a Ph.D. would mean a better salary but I forgot to calibrate for faculty jobs), but somehow I made the right choice—I could not be more grateful for the memorable six years at Cornell and Cornell Tech.

First and foremost, I have been incredibly lucky to have Ari Juels as my Ph.D. advisor. His constant, patient, constructive advice and guidance led me to who I am. I have learned so much—from how to articulate and present ideas, to how to cultivate a research vision, and to many things outside research. Above all, I am especially grateful for his initiatives and guidance on transferring research to real-world adoption. Those experiences not only helped me achieve broader impact, but also reinforced my belief in academia as a venue to achieve meaningful real-world impact, which is the ultimate reason for me to stay in academia. Ari has been and will continue to be my role model as a researcher, advisor, and colleague.

I have also been extremely fortunate to have Elaine Shi, Tom Ristenpart, and Rafael Pass as my thesis committee members. Working with them and being exposed to ideas and perspectives from their areas of expertise have played a central role in the development of my research interests and values. They have always been great supporters throughout my Ph.D. study.

This thesis would not have been possible without my co-authors and colleagues who directly or indirectly contributed to this research. The list is long and I will surely miss many who made an indispensable contribution. Nevertheless, I would like to thank my external collaborators: Andrew Low, Dawn, Jernej, Raymond, Lun (Berkeley), Andrew Miller (UIUC), Florian (Stanford),

Mike (UNC), Ahmed, Yupeng (UMD), as well as my colleagues and friends at Cornell: Andreas, Antonio, Bijeeta, Congzhen Deepak, Ethan, Fabian, Ge, Ian, Iddo, Ittay, Jasleen, Julia, Kyle, Lei, Lorenz, Longqi, Lucy, Mahimna, Nirvan, Paul, Phil, Steven, Tyler, Xiao, Yan, Yin, Yiqing, Yuhang Ma, Yuhang Zhao, and others for making my Ph.D. experience memorable.

My Ph.D. experience has been made unique by Cornell Tech with its beautiful, quiet campus just across the East River from Manhattan, and the brilliance and diversity of its residents, which is also why I love New York City. Thanks for founding Cornell Tech!

Last but not least, I would like to thank my family. My parents have provided unconditional love and support throughout my life and have always encouraged me to be brave to challenge myself and to make a positive impact on the world. I want to especially thank my wife, Zhen Lian, who has been my best friend, closest colleague (she is also a Ph.D. student at Cornell Tech), and occasionally best therapist. I want to thank her for always having a clear and logical mind, for helping me navigate through many complex situations, and for urging me to adopt a healthier lifestyle and many more. Her support and help are essential to all my success. Finally, my cat has provided more soothing power than any two-legged objects.

TABLE OF CONTENTS

| | |
|--|-----------|
| Biographical Sketch | iii |
| Dedication | iv |
| Acknowledgements | v |
| Table of Contents | vii |
| List of Tables | ix |
| List of Figures | x |
| 1 Introduction | 1 |
| 1.1 Protocols for authenticated data oracles | 3 |
| 1.2 Protocols for decentralized secret storage | 6 |
| 1.3 Protocols for resource-efficient mining | 7 |
| 1.4 Thesis organization | 8 |
| 2 Town Crier: An Authenticated Data Feed for Smart Contracts | 9 |
| 2.1 Introduction | 9 |
| 2.2 Background | 15 |
| 2.3 Architecture and Security Model | 18 |
| 2.4 TC Protocol Overview | 22 |
| 2.5 Two Key Security Properties | 26 |
| 2.6 Town Crier Protocol | 32 |
| 2.7 TC Implementation Details | 38 |
| 2.8 Security Analysis | 43 |
| 2.9 Experiments | 45 |
| 2.10 Related Work | 52 |
| 2.11 Future Work | 52 |
| 2.12 Conclusion | 55 |
| 3 DECO: Liberating Web Data Using Decentralized Oracles for TLS | 56 |
| 3.1 Introduction | 56 |
| 3.2 Background | 62 |
| 3.3 Overview | 64 |
| 3.4 The DECO protocol | 76 |
| 3.5 Proof generation | 88 |
| 3.6 Applications | 96 |
| 3.7 Implementation and Evaluation | 103 |
| 3.8 Legal and Compliance Issues | 108 |
| 3.9 Related Work | 109 |
| 3.10 Conclusion | 111 |

| | | |
|----------|---|------------|
| 4 | CHURP: Dynamic-Committee Proactive Secret Sharing | 112 |
| 4.1 | Introduction | 112 |
| 4.2 | Model and Assumptions | 118 |
| 4.3 | Overview of CHURP | 123 |
| 4.4 | Efficient Bivariate 0-Sharing | 129 |
| 4.5 | CHURP Protocol Details | 132 |
| 4.6 | CHURP Implementation & Evaluation | 148 |
| 4.7 | Point-to-Point Technique Details | 155 |
| 4.8 | Applications in Decentralized Systems | 161 |
| 4.9 | Related Work | 164 |
| 5 | REM: Resource-Efficient Mining for Blockchains | 168 |
| 5.1 | Introduction | 168 |
| 5.2 | Background | 173 |
| 5.3 | Overview of PoUW and REM | 177 |
| 5.4 | Tolerating Compromised SGX Nodes | 181 |
| 5.5 | Implementation Details | 191 |
| 5.6 | Waste Analysis | 203 |
| 5.7 | Related Work | 226 |
| 5.8 | Conclusion | 230 |
| 6 | Conclusion and future directions | 232 |
| A | Town Crier | 236 |
| A.1 | Formal Modeling | 236 |
| A.2 | Proofs of Security | 239 |
| A.3 | Applications and Code Samples | 245 |
| B | DECO | 252 |
| B.1 | Protocols details for GCM | 252 |
| B.2 | Security proofs | 255 |
| B.3 | Details on Key-Value Grammars and Two-Stage Parsing | 261 |
| C | CHURP | 267 |
| C.1 | Security Proof for Opt – CHURP | 267 |
| C.2 | CHURP Pessimistic paths | 270 |
| C.3 | The Static Setting: Improved PSS | 275 |
| C.4 | Changing the threshold | 278 |
| D | REM | 281 |
| D.1 | Tolerating Compromised SGX Nodes: Details | 281 |
| | Bibliography | 288 |

LIST OF TABLES

| | | |
|-----|---|-----|
| 2.1 | TC enclave response time. | 47 |
| 3.1 | Run time of DECO protocols. | 104 |
| 3.2 | Run time of the proof-generation phase. | 106 |
| 4.1 | Notation used in CHURP. | 136 |
| 4.2 | Summary of protocol performance. | 145 |
| 4.3 | Performance comparison of on-chain communication methods. | 155 |
| 4.4 | Comparison of Proactive Secret Sharing (PSS) schemes | 165 |

LIST OF FIGURES

| | | |
|------|---|-----|
| 2.1 | Basic Town Crier architecture. | 19 |
| 2.2 | Data flows in datagram processing. | 24 |
| 2.3 | Formal abstraction for SGX execution. | 25 |
| 2.4 | Hybrid TCB in TC. | 28 |
| 2.5 | Offline verification of SGX attestation. | 31 |
| 2.6 | TC contract \mathcal{C}_{TC} reflecting fees. | 34 |
| 2.7 | The Town Crier Relay \mathcal{R} | 35 |
| 2.8 | The Town Crier Enclave $\text{prog}_{\text{encl}}$ | 36 |
| 2.9 | Money Flow for a Delivered Request. | 37 |
| 2.10 | Components of TC Server. | 40 |
| 2.11 | Throughput on a single SGX machine. | 49 |
| | | |
| 3.1 | The oracle functionality. | 67 |
| 3.2 | An overview of the workflow in DECO. | 71 |
| 3.3 | Demonstration of context-integrity attacks. | 75 |
| 3.4 | The protocol of three-party handshake. | 77 |
| 3.6 | The 2PC-HMAC protocol. | 83 |
| 3.7 | The DECO protocol. | 87 |
| 3.8 | An example of confidential binary option using DECO. | 99 |
| 3.9 | An example of age proof. | 102 |
| 3.10 | An example of verifiable claims of price discrimination. | 103 |
| | | |
| 4.1 | An illustration of a handoff between two committees. | 118 |
| 4.2 | An illustration of an epoch. | 119 |
| 4.3 | An example of the handoff protocol. | 127 |
| 4.4 | The specification of $(2t, 2t + 1)$ -UnivariateZeroShare. | 130 |
| 4.5 | The specification of (t, n) -BivariateZeroShare. | 131 |
| 4.6 | CHURP protocol tiers. | 135 |
| 4.7 | Opt-ShareReduce between the committees \mathcal{C} and \mathcal{C}' | 140 |
| 4.8 | Opt-Proactivize updates the reduced shares. | 143 |
| 4.9 | The specification of Opt-ShareDist. | 144 |
| 4.10 | Latency | 151 |
| 4.11 | Off-chain communication overhead. | 152 |
| 4.12 | Tradeoff in latency vs. message transmission cost. | 161 |
| | | |
| 5.1 | Architecture overview of REM. | 181 |
| 5.6 | Miner loop. | 192 |
| 5.7 | REM toolchain. | 194 |
| 5.8 | PoUW Runtime. | 195 |
| 5.11 | REM block structure. | 201 |
| 5.12 | REM Overhead. | 202 |
| 5.14 | Summary of revenue analysis result. | 212 |

CHAPTER 1

INTRODUCTION

One of the most fundamental challenge in computer science is to implement *fault-tolerant services*, namely systems that remain functional and secure even some participants in the system behave maliciously. Towards this goal, Byzantine Fault Tolerance (BFT) [173] has been studied for decades. However, traditional BFT-based protocols only work in a *closed* setting where the number of participants in the system, as well as their identities, are known. This requirement is quite limiting, and in particular these protocols cannot be used in a open and permissionless setting where participants are free to join and leave at any time. How to achieve strong fault-tolerance in a permissionless setting remains unknown, until Nakamoto’s now celebrated *blockchain* protocol [190].

Blockchains realize a simple but powerful abstraction: a public append-only ledger that functions in a fully “decentralized” fashion ¹. This new capability gives rise to an array of new applications. Most excitingly, blockchains enable *smart contracts* (e.g., Ethereum [64]), which are stateful programs executed by the blockchain network. Smart contracts can realize arbitrarily complex logic, and they inherit strong security guarantees from the underlying blockchain, such as transparency, tamper-resistance, and censorship-resistance. In other words, smart contracts are a general method to implement fault-tolerant services in a decentralized setting. These nice properties make smart contracts an appealing solution to a wide range of real-world challenges. Proponents predict that smart contracts will transform areas as diverse as finance, supply chain management,

¹Unfortunately the term “decentralization” has been heavily overloaded and has inconsistent and vague meanings, e.g., as argued in [245]. Throughout this thesis, we use the term to denote that a given system or protocol does not rely on a central authority.

health care, and many more [122, 52, 240].

However, despite the widespread interest and significant investment (e.g., more than \$3.8 billion in VC investment [102] has been made to blockchain startups as of 2019), most of their real-world uses, such as tokens, exercise little of smart contracts' potential power. A key reason is that many envisioned applications demand fundamentally more than what a blockchain can offer (recall that a blockchain is *merely a public append-only ledger*). For example, envisioned applications as diverse as financial contracts, decentralized identity management, and even betting require reliable access to authenticated data about the real world, but this is currently no secure, decentralized way to establish authenticated channels between smart contracts and existing off-chain data sources (e.g., websites). Moreover, while smart contracts inherit the strong security guarantees from the underlying blockchains, they also inherit fundamental limitations: they cannot provide any privacy and suffer from limited performance.

A promising direction to overcome these limitations is to combine on-chain computation with off-chain protocols (e.g., multiparty computation (MPC) and trusted execution environments (TEE)). However, how to design, model, and analyze heterogeneous systems that span blockchains and off-chain components with *disparate security properties* is the main technical challenge this thesis tackled. **This thesis explores principled composition of on-chain and off-chain components that lead to provably secure, efficient, and decentralized systems.**

This thesis advances the state of the art of smart contract capabilities by filling in basic yet missing blockchain needs. Specifically, this thesis explores three research directions: (1) **authenticated data oracles** that enable the faithful representation of real-world states on blockchains. We present Town Crier and

DECO, two systems that can convert widely deployed TLS-enabled data sources to authenticated data feeds. (2) **decentralized secret storage** that endows smart contracts with privacy by storing secrets and performing computation in off-chain committees. We present CHRUP, the first practical churn-robust secret-sharing protocol that achieves orders-of-magnitude performant improvement over prior works. (3) **resource-efficient consensus** by Proof of Useful Work. We present REM, a system that leverage trusted execution environments (TEEs) to realize PoW-like consensus protocol while avoiding wasteful computation.

Industry adoption. Beyond the scholarly contributions, several systems in this thesis has been adopted commercially. Town Crier (Chapter 2) and DECO (Chapter 3) have been licensed by ChainLink (<https://chain.link>) to build decentralized oracle networks. CHURP (Chapter 4) is on the product roadmap of Oasis Labs (<https://www.oasislabs.com>) as a potential key management solution.

The following are further summaries of our contributions.

1.1 Protocols for authenticated data oracles

Many envisioned smart contract use cases (e.g., financial instruments) require them to consume off-chain data (e.g., stock quotes). *External data must be pushed to the blockchain by an oracle before they can be used by a smart contract.* Oracles are of vital importance to the trustworthiness of smart contracts—if they can be tampered with, blockchains’ guarantees of tamper-resistance is meaningless; if they’re controlled by a central party, blockchain’s decentralization promise is negated. Moreover, oracles are natural attack targets. A recent exploit [197] of er-

rors in the price oracle used by Synthetix (a decentralized exchange) allowed the hacker to gain 1,000x profits equalling \$1 billion in less than an hour. Therefore, trustworthy oracles are a cornerstone to the smart contract ecosystem and a key missing component to realize the envisioned applications.

The main challenge in building oracles is to achieve strong authenticity and privacy guarantees while maintaining backward compatibility with existing data sources. Ample authenticated data are already readily available through secure (i.e., HTTPS) websites and APIs, but deployed secure protocols—Transport Layer Security (TLS)—is incompatible with how oracles are supposed to work, as TLS does *not* guarantee authenticity for relayed data. As a result, existing oracle designs either cannot prove authenticity (i.e., one must trust the oracle for that) or require changing how millions of deployed websites release data, which will incur a high adoption cost.

The first two chapters of this thesis present protocols that address this challenge. While two systems explore different tradeoffs, they both enable smart contracts to securely use existing TLS-enabled data sources *without any modifications to existing data sources*.

1.1.1 Town Crier (TC)

TC draws upon recent advances in trusted computing: Intel SGX [185, 32, 146] provides a CPU-based trusted execution environment (TEE) that aims to provide integrity and confidentiality guarantees to computation performed on a computer where all the privileged software (e.g., kernel) is potentially malicious.

TC has a hybrid architecture with a smart contract front-end and a TEE backend. Essentially, the TC backend terminates TLS sessions in a TEE and signs the payload using a secret key generated in the TEE.

TC is one of the *first* systems that employ a *hybrid* trusted computing base (TCB) that spans smart contracts and TEEs, which introduced modeling and design challenges. One of our contributions is a formal model of TC that captures not only traditional security properties (i.e., authenticity), but also economic safety properties (e.g., TC will not run out of money and that an honest user will not pay excessive fees) uniquely induced by the monetary cost associated with smart contract execution. We also proposed techniques to minimize the size of hybrid TCBs that can be applied to other TEE-blockchain systems as well.

Town Crier also supports confidentiality. It enables private data requests with encrypted parameters. Additionally, in a generalization that executes smart-contract logic within Town Crier, the system permits secure use of user credentials to scrape access-controlled online data sources.

1.1.2 DECO

The use of TEEs in TC enables performant and powerful features that are otherwise unachievable (e.g., complex post-processing of TLS data). Nonetheless, it also introduces reliance on TEE security that may be undesirable for some applications.

To this end, we explored a different trust model without TEEs and developed DECO, a three-party cryptographic protocol between a user, an oracle, and a TLS

server. DECO allows the user to prove facts about her TLS sessions (with the server) to the oracle in a privacy-preserving fashion.

DECO draws on latest advances in Multiparty Computation (MPC) and zero-knowledge proofs (ZKPs). The novelty of DECO lies in achieving practical performance while not requiring any changes on the server side. To this end, DECO employs several novel cryptographic techniques. DECO introduces a three-party TLS handshake protocol that secret-share the session keys between the oracle and user, as well as highly optimized zero-knowledge proofs that allow the user to prove fine-grained statements about her session data efficiently.

1.2 Protocols for decentralized secret storage

Blockchains can't store secrets, which deprives smart contracts of privacy and, as a consequence, the ability to use cryptography—e.g., a smart contract can't issue digital signatures because the blockchain can't store the secret signing key.

One idea is to delegate the storage of secrets to *off-chain committees* using Secret Sharing (SS) [229], or better yet, Proactive Secret Sharing (PSS) [145]. However, most of the existing PSS protocols assume a *static committee* that runs against the open-membership nature of blockchains. In permissionless blockchains, nodes may freely join and leave the system at any time. In permissioned blockchains, only authorized nodes can join, but nodes can fail and membership change. Either way, secret sharing for blockchains must support committee membership changes, i.e., *dynamic committees*. Under node churn, existing PSS is either insecure (i.e., leaks the secret to the adversary) or incur prohibitive communication costs.

In this thesis, we present CHURP, a dynamic-committee PSS with a *very low communication complexity*. CHURP employed several novel secret-sharing techniques to reduce communication complexity: a new zero-sharing protocol for efficient proactivization, dimension-switching to safeguard the secret in committee handoffs, and hedging techniques for trusted setup failures. CHURP achieves an $O(n^2)$ improvement of communication complexity over the state of the art [226], and concretely at least 1000x less communication overhead for a committee of hundreds of nodes.

CHURP enables a wide range of applications requiring maintaining secrets in a decentralized system, such as decentralized key management, decentralized identity, auditable access control, committee-based consensus, and decentralized secure multiparty computation (MPC).

1.3 Protocols for resource-efficient mining

One of the core blockchain innovations is the first *permissionless* consensus protocol, known as the Nakamoto consensus [190]. It uses cryptographic puzzles (solutions to which are called Proofs of Work or PoW) to prevent Sybil attacks. The cost, however, is the computation wasted searching for PoWs (aka mining), which serve no useful purpose beyond consensus. The arm race in mining has led to enormous monetary and environmental costs: the Bitcoin network consumes as much electricity as the entire Israel as of May, 2020 [103].

Can we design puzzles differently such that their solutions are useful? To this end, I developed Proofs of Useful Work, or PoUW. To search for a PoUW, miners perform useful computation (e.g., scientific experiments, pharmaceuti-

cal discovery) in TEEs. Each unit of work done grants the miner a small and dynamically calibrated probability to discover a PoUW. The design challenge is to find fair, robust, and efficient ways to measure the amount of work done in TEEs. I designed efficient binary analysis algorithms that can reliably count CPU instructions while TEE programs execute. The rest of the protocol is essentially the same as Nakamoto's. Therefore PoUW inherits all the desired properties of PoW but avoids energy waste.

1.4 Thesis organization

The rest of this thesis is organized in five chapters covering works from [257, 260, 182, 259] done in collaboration with Ethan Cecchetti, Kyle Croman, Robert Escriva, Ittay Eyal, Steven Goldfeder, Ari Juels, Andrew Low, Harjasleen Malvai, Sai Krishna Deepak Maram, Robbert Van Renesse, Elaine Shi, Dawn Song, Lun Wang, and Yupeng Zhang.

Chapter 2 and Chapter 3 presents Town Crier and DECO, respectively, two data oracle protocols for TLS with different trust assumptions and performance profiles. Chapter 4 presents CHURP, a protocol for churn-robust secret sharing. Chapter 5 presents REM and its Proof of Useful Work (PoUW) protocol that achieve a PoW-style consensus without wasting computation power. Lastly, Chapter 6 concludes the thesis with a discussion of future research directions.

CHAPTER 2
TOWN CRIER: AN AUTHENTICATED DATA FEED FOR SMART
CONTRACTS

2.1 Introduction

Smart contracts are computer programs that autonomously execute the terms of a contract. For decades they have been envisioned as a way to render legal agreements more precise, pervasive, and efficiently executable. Szabo, who popularized the term “smart contract” in a seminal 1994 essay [238], gave as an example a smart contract that enforces car loan payments. If the owner of the car fails to make a timely payment, a smart contract could programmatically revoke physical access and return control of the car to the bank.

Cryptocurrencies such as Bitcoin [190] provide key technical underpinnings for smart contracts: direct control of money by programs and fair, automated code execution through the decentralized consensus mechanisms underlying blockchains. The recently launched Ethereum [250, 64] supports Turing-complete code and thus fully expressive self-enforcing decentralized smart contracts—a big step toward the vision of researchers and proponents. As Szabo’s example shows, however, the most compelling applications of smart contracts—such as financial instruments—additionally require access to *data about real-world state and events*.

Data feeds (also known as “oracles”) aim to meet this need. Very simply, data feeds are contracts on the blockchain that serve data requests by other contracts [64, 250]. A few data feeds exist for Ethereum today that source data

from trustworthy websites, but provide no assurance of correctly relaying such data beyond the reputation of their operators (typically individuals or small entities). HTTPS connection to a trustworthy website would seem to offer a solution, but smart contracts lack network access, and HTTPS does not digitally sign data for out-of-band verification. The lack of a substantive ecosystem of trustworthy data feeds is frequently cited as critical obstacle to the evolution of Ethereum and decentralized smart contracts in general [135].

Town Crier. We introduce a system called *Town Crier* (TC) that addresses this challenge by providing an *authenticated data feed* (ADF) for smart contracts. TC acts as a high-trust bridge between existing HTTPS-enabled data websites and the Ethereum blockchain. It retrieves website data and serves it to relying contracts on the blockchain as concise pieces of data (e.g. stock quotes) called *datagrams*. TC uses a novel combination of Software Guard Extensions (SGX), Intel’s recently released trusted hardware capability, and a smart-contract front end. It executes its core functionality as a trusted piece of code in an SGX *enclave*, which protects against malicious processes and the OS and can *attest* (prove) to a remote client that the client is interacting with a legitimate, SGX-backed instance of the TC code.

The smart-contract front end of Town Crier responds to requests by contracts on the blockchain with attestations of the following form:

“Datagram X specified by parameters $params$ is served by an HTTPS-enabled website Y during a specified time frame T .”

A relying contract can verify the correctness of X in such a datagram assuming

trust only in the security of SGX, the (published) TC code, and the validity of source data in the specified interval of time.

Another critical barrier to smart contract adoption is the lack of *confidentiality* in today's ecosystems; all blockchain state is publicly visible, and existing data feeds publicly expose requests. TC provides confidentiality by supporting *private* datagram requests, in which the parameters are encrypted under a TC public key for ingestion in TC's SGX enclave and are therefore concealed on the blockchain. TC also supports *custom* datagram requests, which securely access the online resources of requesters (e.g. online accounts) by ingesting encrypted user credentials, permitting TC to securely retrieve access-controlled data.

We designed and implemented TC as a complete, highly scalable, end-to-end system that offers formal security guarantees at the cryptographic protocol level. TC runs on real, SGX-enabled host, as opposed to an emulator (e.g. [46, 227]). We plan to launch a version of TC as an open-source, production service atop Ethereum, pending the near-future availability of the Intel Attestation Service (IAS), which is needed to verify SGX attestations.

Technical challenges. Smart contracts execute in an adversarial environment where parties can reap financial gains by subverting the contracts or services on which they rely. Formal security is thus vitally important. We adopt a rigorous approach to the design of Town Crier by modeling it in the Universal Composability (UC) framework, building on [171, 230] to achieve an interesting formal model that spans a blockchain and trusted hardware. We formally define and prove that TC achieves the basic property of datagram *authenticity*—informally that TC faithfully relays current data from a target website. We additionally

prove *fair expenditure* for an honest requester, informally that the fee paid by a user contract calling TC is at most a small amount to cover the operating costs of the TC service, even if the TC host is malicious.

Another contribution of our work is introducing and showing how to achieve two key security properties: *gas sustainability* and *trusted computing base (TCB) code minimization* within a new TCB model created by TC's combination of a blockchain with SGX.

Because of the high resource costs of decentralized code execution and risk of application-layer denial-of-service (DoS) attacks, Ethereum includes an accounting resource called *gas* to pay for execution costs. Informally, *gas sustainability* means that an Ethereum service never runs out of gas, a general and fundamental availability property. We give a formal definition of gas sustainability applicable to any Ethereum service, and prove that TC satisfies it.

We believe that the combination of blockchains with SGX introduced in our work will prove to be a powerful and general way to achieve confidentiality in smart contract systems and network them with off-chain systems. This new security paradigm, however, introduces a hybridized TCB that spans components with different trust models. We introduce techniques for using such a hybridized TCB securely while *minimizing the TCB code size*. In TC, we show how to avoid constructing an authenticated channel from the blockchain to the enclave—bloating the enclave with an Ethereum client—by instead authenticating enclave outputs on the blockchain. We also show how to minimize on-chain signature-verification code. These techniques are general; they apply to any use of a similar hybridized TCB.

Other interesting smaller challenges arise in the design of TC. One is deployment of TLS in an enclave. Enclaves lack networking capabilities, so TLS code must be carefully partitioned between the enclave and untrusted host environment. Another is hedging in TC against the risk of compromise of a website or single SGX instance, which we accomplish with various modes of majority voting: among multiple websites offering the same piece of data (e.g. stock price) or among multiple SGX platforms.

Applications and performance. We believe that TC can spur deployment of a rich spectrum of smart contracts that are hard to realize in the existing Ethereum ecosystem. We explore three examples that demonstrate TC’s capabilities: (1) A financial derivative (cash-settled put option) that consumes stock ticker data; (2) A flight insurance contract that relies on private data requests about flight cancellations; and (3) A contract for sale of virtual goods and online games (via Steam Marketplace) for Ether, the Ethereum currency, using custom data requests to access user accounts.

Our experiments with these three applications show that TC is highly scalable. Running on just a single SGX host, TC achieves throughputs of 15-65 tx/sec. TC is easily parallelized across many hosts, as separate TC hosts can serve requests with no interdependency. (For comparison, Ethereum handles less than 1 tx/sec today and recent work [91] suggests that Bitcoin can scale safely to no more than 26 tx/sec with reparametrization.) For these same applications, experimental response times for datagram requests range from 192-1309 ms—much less than an Ethereum block interval (12 seconds on average). These results suggest that a few SGX-enabled hosts can support TC data feed rates well beyond the global transaction rate of a modern decentralized blockchain.

Contributions. We offer the following contributions:

- We introduce and report on an end-to-end implementation of Town Crier, an authenticated data feed system that addresses critical barriers to the adoption of decentralized smart contracts. TC combines a smart-contract front end in Ethereum and an SGX-based trusted hardware back end to: (1) Serve authenticated data to smart contracts without a trusted service operator and (2) Support *private* and *custom* data requests, enabling encrypted requests and secure use of access-controlled, off-chain data sources. We plan to launch a version of TC soon as an open-source service.
- We formally analyze the security of TC within the Universal Composability (UC) framework, defining functionalities to represent both on-chain and off-chain components. We formally define and prove the basic properties of datagram *authenticity* and *fair expenditure* as well as *gas sustainability*, a fundamental availability property for any Ethereum service.
- We introduce a hybridized TCB spanning the blockchain and an SGX enclave, a powerful new paradigm of trustworthy system composition. We present generic techniques that help shrink the TCB code size within this model as well as techniques to hedge against individual SGX platform compromises.
- We explore three TC applications that show TC's ability to support a rich range of services well beyond those in Ethereum today. Experiments with these applications also show that TC can easily meet the latency and throughput requirements of modern decentralized blockchains.

2.2 Background

In this section, we provide basic background on the main technologies TC incorporates, namely SGX, TLS / HTTPS, and smart contracts.

SGX. Intel’s Software Guard Extensions (SGX) [185, 32, 146] is a set of new instructions that confer hardware protections on user-level code. SGX enables process execution in a protected address space known as an *enclave*. The enclave protects the confidentiality and integrity of the process from certain forms of hardware attack and other software on the same host, including the operating system.

An enclave process cannot make system calls, but can read and write memory outside the enclave region. Thus isolated execution in SGX may be viewed in terms of an ideal model in which a process is guaranteed to execute correctly and with perfect confidentiality, but relies on a (potentially malicious) operating system for network and file-system access.¹

SGX allows a remote system to verify the software in an enclave and communicate securely with it. When an enclave is created, the CPU produces a hash of its initial state known as a *measurement*. The software in the enclave may, at a later time, request a report which includes a measurement and supplementary data provided by the process, such as a public key. The report is digitally signed using a hardware-protected key to produce a proof that the measured software is running in an SGX-protected enclave. This proof, known as a *quote*, can be ver-

¹This model is a simplification: SGX is known to expose some internal enclave state to the OS [88]. Our basic security model for TC assumes ideal isolated execution, but again, TC can also be distributed across multiple SGX instances as a hedge against compromise.

ified by a remote system, while the process-provided public key can be used by the remote system to establish a secure channel with the enclave or verify signed data it emits. We use the generic term *attestation* to refer to a quote, and denote it by *att*. We assume that a trustworthy measurement of the code for the enclave component of TC is available to any client that wishes to verify an attestation. SGX signs quotes using a *group signature* scheme called EPID [58]. This choice of primitive is significant in our design of Town Crier, as EPID is a proprietary signature scheme not supported natively in Ethereum. SGX additionally provides a trusted time source via the function `sgx_get_trusted_time`. On invoking this function, an enclave obtains a measure of time relative to a reference point indexed by a nonce. A reference point remains stable, but SGX does not provide a source of absolute or wall-clock time, another limitation we must work around in TC.

TLS / HTTPS. We assume basic familiarity by readers with TLS and HTTPS (HTTP over TLS). As we explain later, TC exploits an important feature of HTTPS, namely that it can be partitioned into interoperable layers: an HTTP layer interacting with web servers, a TLS layer handling handshakes and secure communication, and a TCP layer providing reliable data stream.

Smart contracts. While TC can in principle support any smart-contract system, we focus in this paper on its use in Ethereum, whose model we now explain. For further details, see [64, 250].

A smart contract in Ethereum is represented as what is called a *contract account*, endowed with code, a currency balance, and persistent memory in the form of a key/value store. A contract accepts messages as inputs to any

of a number of designated functions. These entry points, determined by the contract creator, represent the API of the contract. Once created, a contract executes autonomously; it persists indefinitely with even its creator unable to modify its code.² Contract code executes in response to receipt of a *message* from another contract or a *transaction* from a non-contract (*externally owned*) account, informally what we call a *wallet*. Thus, contract execution is always initiated by a transaction. Informally, a contract only executes when “poked,” and poking progresses through a sequence of entry points until no further message passing occurs (or a shortfall in gas occurs, as explained below). The “poking” model aside, as a simple abstraction, a smart contract may be viewed as an *autonomous agent* on the blockchain.

Ethereum has its own associated cryptocurrency called *Ether*. To prevent DoS attacks, prevent inadvertent infinite looping within contracts, and generally control network resource expenditure, Ethereum allows Ether-based purchase of a resource called *gas* to power contracts. Every operation, including sending data, executing computation, and storing data, has a fixed gas cost. Transactions include a parameter (GASLIMIT) specifying a bound on the amount of gas expended by the computations they initiate. When a function calls another function, it may optionally specify a lower GASLIMIT for the child call which expends gas from the same pool as the parent. Should a function fail to complete due to a gas shortfall, it is aborted and any state changes induced by the partial computation are rolled back to their pre-call state; previous computations on the call path, though, are retained and gas is still spent.

Along with a GASLIMIT, a transaction specifies a GASPRICE, the maximum amount in Ether that the transaction is willing to pay per unit of gas. The

²There is one exception: a special opcode `suicide` wipes code from a contract account.

transaction thus succeeds only if the initiating account has a balance of $\text{GASLIMIT} \times \text{GASPRICE}$ Ether and GASPRICE is high enough to be accepted by the system (miner).

As we discuss in Section 2.5.1, the management of gas is critical to the availability of TC (and other Ethereum-based services) in the face of malicious users.

Finally, we note that transactions in Ethereum are digitally signed for a wallet using ECDSA on the curve Secp256k1 and the hash function SHA3-256 .

2.3 Architecture and Security Model

Town Crier includes three main components: The TC Contract (\mathcal{C}_{TC}), the Enclave (whose code is denoted by $\text{prog}_{\text{encl}}$), and the Relay (\mathcal{R}). The Enclave and Relay reside on the TC server, while the TC Contract resides on the blockchain. We refer to a smart contract making use of the Town Crier service as a *requester* or *relying* contract, which we denote \mathcal{C}_U , and its (off-chain) owner as a *client* or *user*. A *data source*, or *source* for short, is an online server (running HTTPS) that provides data which TC draws on to compose datagrams.

An architectural schematic of TC showing its interaction with external entities is given in Fig. 2.1.

The TC Contract \mathcal{C}_{TC} . The TC Contract is a smart contract that acts as the blockchain front end for TC. It is designed to present a simple API to a relying contract \mathcal{C}_U for its requests to TC. \mathcal{C}_{TC} accepts datagram requests from \mathcal{C}_U and returns corresponding datagrams from TC. Additionally, \mathcal{C}_{TC} manages TC's

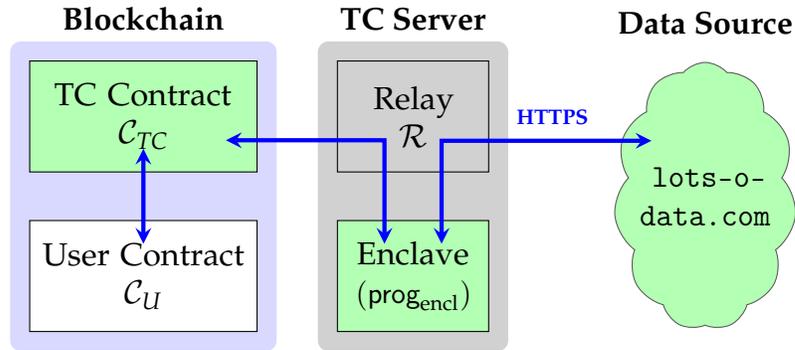


Figure 2.1: Basic Town Crier architecture. Trusted components are depicted in green.

monetary resources.

The Enclave. We refer to an instance of the TC code running in an SGX enclave simply as the Enclave and denote the code itself by $\text{prog}_{\text{encl}}$. In TC, the Enclave ingests and fulfills datagram requests from the blockchain. To obtain the data for inclusion in datagrams, it queries external data sources, specifically HTTPS-enabled internet services. It returns a datagram to a requesting contract \mathcal{C}_U as a digitally signed blockchain message. Under our basic security model for SGX, network functions aside, the Enclave runs in complete isolation from an adversarial OS as well as other process on the host.

The Relay \mathcal{R} . As an SGX enclave process, the Enclave lacks direct network access. Thus the Relay handles bidirectional network traffic on behalf of the Enclave. Specifically, the Relay provides network connectivity from the Enclave to three different types of entities:

1. *The Blockchain (the Ethereum system):* The Relay scrapes the blockchain in order to monitor the state of \mathcal{C}_{TC} . In this way, it performs implicit message

passing from C_{TC} to the Enclave, as neither component itself has network connectivity. Additionally, the Relay places messages emitted from the Enclave (datagrams) on the blockchain.

2. *Clients*: The Relay runs a web server to handle off-chain service requests from clients—specifically requests for Enclave attestations. As we soon explain, an attestation provides a unique public key for the Enclave instance to the client and proves that the Enclave is executing correct code in an SGX enclave and that its clock is correct in terms of absolute (wall-clock) time. A client that successfully verifies an attestation can then safely create a relying contract C_U that uses the TC.
3. *Data sources*: The Relay relays traffic between data sources (HTTPS-enabled websites) and the Enclave.

The Relay is an ordinary user-space application. It does not benefit from integrity protection by SGX and thus, unlike the Enclave, can be subverted by an adversarial OS on the TC server to cause delays or failures. A key design aim of TC, however, is that Relay should be unable to cause incorrect datagrams to be produced or users to lose fees paid to TC for datagrams (although they may lose gas used to fuel their requests). As we will show, in general the Relay *can only mount denial-of-service attacks against TC*.

Security model. Here we give a brief overview of our security model for TC, providing more details in later sections. We assume the following:

- *The TC Contract.* C_{TC} is globally visible on the blockchain and its source code is published for clients. Thus we assume that C_{TC} behaves honestly.

- *Data sources.* We assume that clients trust the data sources from which they obtain TC datagrams. We also assume that these sources are stable, i.e., yield consistent datagrams, during a requester’s specified time interval T . (Requests are generally time-invariant, e.g., for a stock price at a particular time.)
- *Enclave security.* We make three assumptions: (1) The Enclave behaves honestly, i.e., $\text{prog}_{\text{encl}}$, whose source code is published for clients, correctly executes the protocol; (2) For an Enclave-generated keypair (sk_{TC}, pk_{TC}) , the private key sk_{TC} is known only to the Enclave; and (3) The Enclave has an accurate (internal) real-time clock. We explain below how we use SGX to achieve these properties.
- *Blockchain communication.* Transaction and message sources are authenticable, i.e., a transaction m sent from wallet \mathcal{W}_X (or message m from contract \mathcal{C}_X) is identified by the receiving account as originating from X . Transactions and messages are integrity protected (as they are digitally signed by the sender), but not confidential.
- *Network communication.* The Relay (and other untrusted components of the TC server) can tamper with or delay communications to and from the Enclave. (As we explain in our SGX security model, the Relay cannot otherwise observe or alter the Enclave’s behavior.) Thus the Relay is subsumed by an adversary that controls the network.

2.4 TC Protocol Overview

We now outline the protocol of TC at a high level. The basic structure is conceptually simple: a user contract \mathcal{C}_U requests a datagram from the TC Contract \mathcal{C}_{TC} , \mathcal{C}_{TC} forwards the request to the Enclave and then returns the response to \mathcal{C}_U . There are many details, however, relating to message contents and protection and the need to connect the off-chain parts of TC with the blockchain.

First we give a brief overview of the protocol structure. Then we enumerate the data flows in TC. Finally, we present the framework for modeling SGX as ideal functionalities inspired by the universal-composability (UC) framework.

2.4.1 Datagram Lifecycle

The lifecycle of a datagram may be briefly summarized in the following steps:

- **Initiate request.** \mathcal{C}_U sends a datagram request to \mathcal{C}_{TC} on the blockchain.
- **Monitor and relay.** The Relay monitors \mathcal{C}_{TC} and relays any incoming datagram request with parameters params to the Enclave.
- **Securely fetch feed.** To process the request specified in params , the Enclave contacts a data source via HTTPS and obtains the requested datagram. It forwards the datagram via the Relay to \mathcal{C}_{TC} .
- **Return datagram.** \mathcal{C}_{TC} returns the datagram to \mathcal{C}_U .

We now make this data flow more precise.

2.4.2 Data Flows

A datagram request by \mathcal{C}_U takes the form of a message $m_1 = (\text{params}, \text{callback})$ to \mathcal{C}_{TC} on the blockchain. params specifies the requested datagram, e.g., $\text{params} := (\text{url}, \text{spec}, T)$, where url is the target data source, spec specifies content of a the datagram to be retrieved (e.g., a stock ticker at a particular time), and T specifies the delivery time for the datagram (initiated by scraping of the data source). The parameter callback in m_1 indicates the entry point to which the datagram is to be returned. While callback need not be in \mathcal{C}_U , we assume it is for simplicity.

\mathcal{C}_{TC} generates a fresh unique id and forwards $m_2 = (\text{id}, \text{params})$ to the Enclave. In response it receives $m_3 = (\text{id}, \text{params}, \text{data})$ from the TC service, where data is the datagram (e.g., the desired stock ticker price). \mathcal{C}_{TC} checks the consistency of params on the request and response and, if they match, forwards data to the callback entry point in message m_4 .

For simplicity here, we assume that \mathcal{C}_U makes a one-time datagram request. Thus it can trivially match m_4 with m_1 . Our full protocol contains an optimization by which \mathcal{C}_{TC} returns id to \mathcal{C}_U after m_1 as a consistent, trustworthy identifier for all data flows. This enables straightforward handling of multiple datagram requests from the same instance of \mathcal{C}_U .

Figure 2.2 shows the data flows involved in processing a datagram request. For simplicity, the figure omits the Relay, which is only responsible for data passing.

Digital signatures are needed to authenticate messages, such as m_3 , entering the blockchain from an external source. We let $(\text{sk}_{TC}, \text{pk}_{TC})$ denote the private / public keypair associated with the Enclave for such message authentication. For

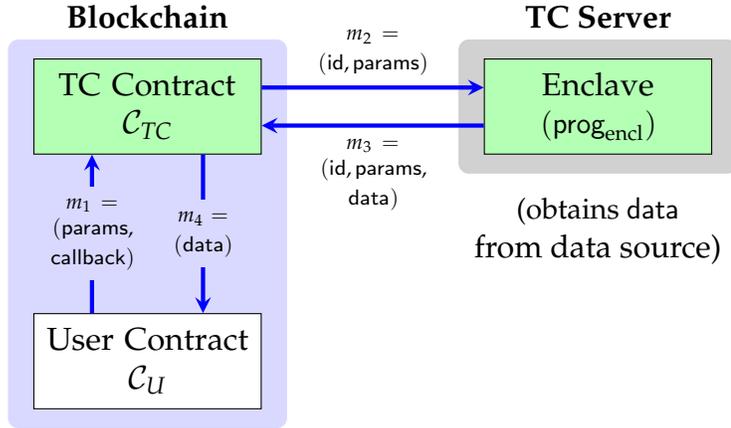


Figure 2.2: Data flows in datagram processing.

simplicity, Fig. 2.2 assumes that the Enclave can send signed messages directly to C_{TC} . We explain later how TC uses a layer of indirection to send m_3 as a transaction via an Ethereum wallet \mathcal{W}_{TC} .

2.4.3 Use of SGX

Let $prog_{encl}$ represent the code for Enclave, which we presume is trusted by all system participants. Our protocols in TC rely on the ability of SGX to attest to execution of an instance of $prog_{encl}$. To achieve this goal, we first present a model that abstracts away the details of SGX, helping to simplify our protocol presentation and security proofs. We also explain how we use the clock in SGX. Our discussion draws on formalism for SGX from Shi et al. [230].

Formal model and notation. We adopt a formal abstraction of Intel SGX proposed by Shi et al. [230]. Following the UC and GUC paradigms [70, 71, 72], Shi et al. propose to abstract away the details of SGX implementation, and instead view SGX as a third party trusted for both confidentiality and integrity. Specifically,

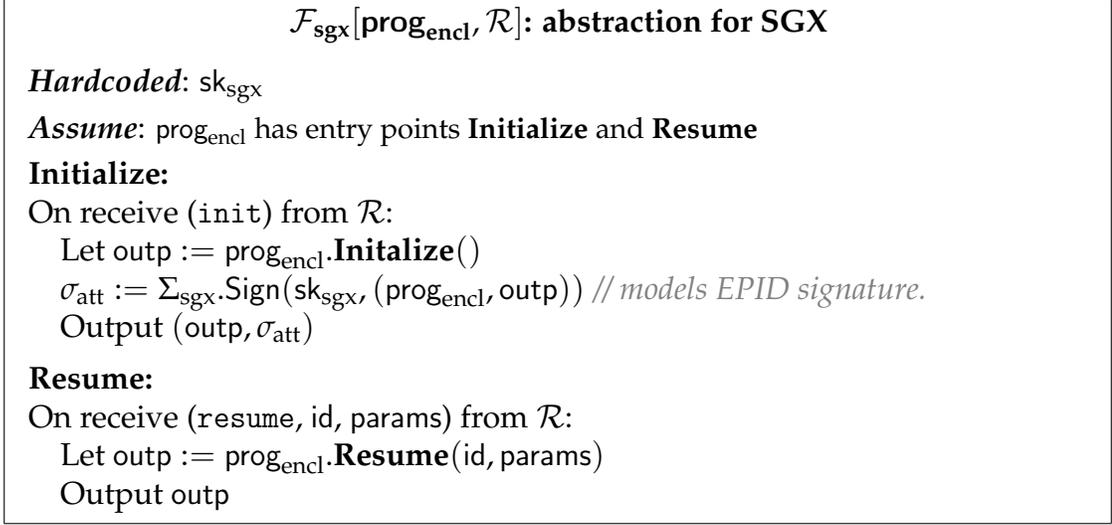


Figure 2.3: Formal abstraction for SGX execution capturing a subset of SGX features sufficient for implementation of TC.

we use a global UC functionality $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})[\text{prog}_{\text{encl}}, \mathcal{R}]$ to denote (an instance of) an SGX functionality parameterized by a (group) signature scheme Σ_{sgx} . Here $\text{prog}_{\text{encl}}$ denotes the SGX enclave program and \mathcal{R} the physical SGX host (which we assume for simplicity is the same as that of the TC Relay). As described in Fig. 2.3, upon initialization, \mathcal{F}_{sgx} runs $\text{outp} := \text{prog}_{\text{encl}}.\mathbf{Initialize}()$ and attests to the code of $\text{prog}_{\text{encl}}$ as well as outp . Upon a resume call with $(\text{id}, \text{params})$, \mathcal{F}_{sgx} runs and outputs the result of $\text{prog}_{\text{encl}}.\mathbf{Resume}(\text{id}, \text{params})$. Further formalism for \mathcal{F}_{sgx} is given Appendix A.1.1.

SGX Clock. As noted above, the trusted clock for SGX provides only relative time with respect to a reference point. To work around this, the Enclave is initialized with the current wall-clock time provided by a trusted source (e.g., the Relay under a trust-on-first-use model). In the current implementation of TC, clients may, in real time, request and verify a fresh timestamp—signed by the Enclave under pk_{TC} —via a web interface in the Relay. Thus, a client can determine the absolute clock time of the Enclave to within the round-trip time of its attestation

request plus the attestation verification time—hundreds of milliseconds in a wide-area network. This high degree of accuracy is potentially useful for some applications but only loose accuracy is required for most. Ethereum targets a block interval of 12s and the clock serves in TC primarily to: (1) Schedule connections to data sources and (2) To check TLS certificates for expiration when establishing HTTPS connections. For simplicity, we assume in our protocol specifications that the Enclave clock provides accurate wall-clock time in the canonical format of seconds since the Unix epoch January 1, 1970 00:00 UTC. Note that the trusted clock for SGX, backed by Intel Manageability Engine [151], is resilient to power outages and reboots [219].

We let $\text{clock}()$ denote measurement of the SGX clock from within the enclave, expressed as the current absolute (wall-clock) time.

2.5 Two Key Security Properties

Before presenting the TC protocol details, we discuss two key security properties informing its design: gas sustainability and TCB minimization in TC’s hybridized TCB model. While we introduce them in this work, as we shall explain, they are of broad and general applicability.

2.5.1 Gas Sustainability

As explained above, Ethereum’s fee model requires that gas costs be paid by the user who initiates a transaction, including all costs resulting from dependent calls. This means that a service that initiates calls to Ethereum contracts must

spend money to execute those calls. Without careful design, such services run the risk of malicious users (or protocol bugs) draining financial resources by triggering blockchain calls for which the service's fees will not be reimbursed. This could cause financial depletion and result in an application-layer denial-of-service attack. It is thus critical for the availability of Ethereum-based services that they always be reimbursed for blockchain computation they initiate.

To ensure that a service is not vulnerable to such attacks, we define *gas sustainability*, a new condition necessary for the liveness of blockchain contract-based services. Gas sustainability is a basic requirement for any self-perpetuating Ethereum service. It can also generalize beyond Ethereum; any decentralized blockchain-based smart contract system must require fees of some kind to reimburse miners for performing and verifying computation.

Let $\text{bal}(\mathcal{W})$ denote the balance of an Ethereum wallet \mathcal{W} .

Definition 2.1 (*K-Gas Sustainability*). *A service with wallet \mathcal{W} and blockchain functions f_1, \dots, f_n is K -gas sustainable if the following holds. If $\text{bal}(\mathcal{W}) \geq K$ prior to execution of any f_i and the service behaves honestly, then after each execution of an f_i initiated by \mathcal{W} , $\text{bal}(\mathcal{W}) \geq K$.*

Recall that a call made in Ethereum with insufficient gas will abort, but spend all provided gas. While Ethereum trivially guarantees 0-gas sustainability, if a transaction is submitted by a wallet with insufficient funds, the wallet's balance will drop to 0. Therefore, to be K -gas sustainable for $K > 0$, each blockchain call made by the service must reimburse gas expenditures. Moreover, the service must have sufficient gas for each call or such reimbursement will be reverted with the rest of the transaction.

The need for gas sustainability (with $K > 0$, as required by TC) informs our protocol design in Section 2.6. We prove that TC achieves this property in Section 2.8.

2.5.2 Hybrid TCB Minimization

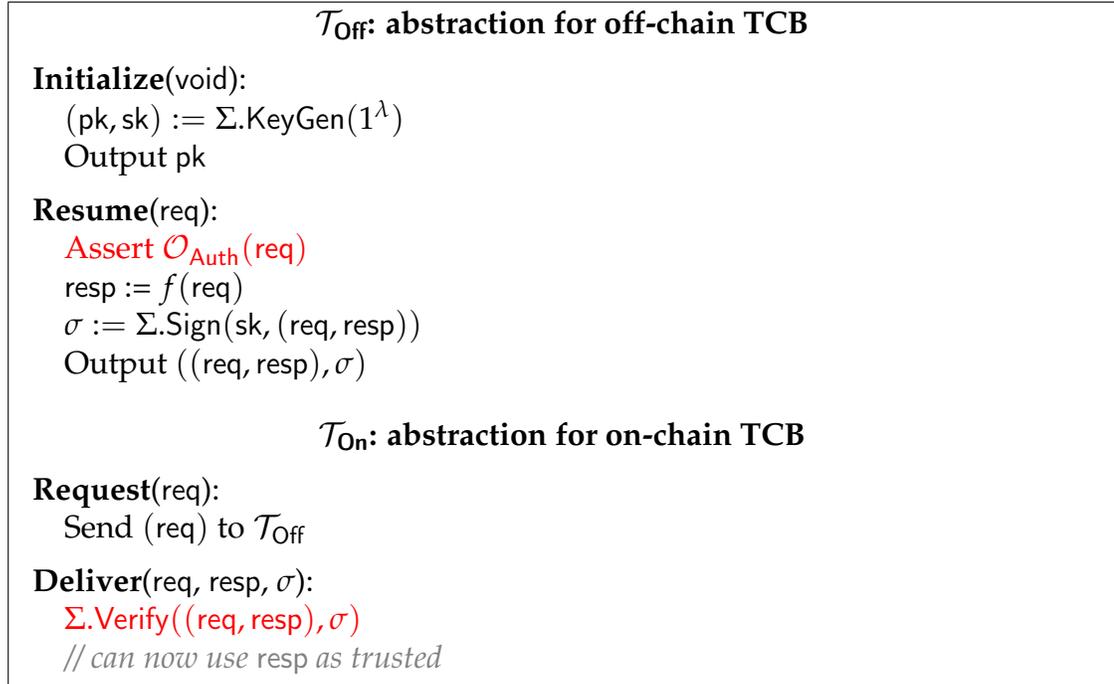


Figure 2.4: Systems like TC have a hybrid TCB. Authentication between two components can greatly increase TCB complexity of implemented naively. We propose techniques to eliminate the most expensive operations (highlighted in red).

In a system involving a smart contract interacting with an off-chain trusted computing environment (e.g. SGX), the TCB is a hybrid of two components with distinct properties. Computation in the smart contract is slow, costly, and completely transparent, meaning it cannot rely on secrets. An SGX enclave is computationally powerful and executes privately, but all external interaction—notably including communication with the contract—must go through an un-

trusted intermediary. While this hybrid TCB is powerful and useful well beyond TC, it presents a challenge: establishing secure communication between the components while minimizing the code in the TCB.

We define abstractions for both TCB components in Fig. 2.4. To distinguish these abstractions from formal ideal functionalities, we use \mathcal{T} (for trusted component), rather than \mathcal{F} . We model the authentication of on-chain messages by an oracle $\mathcal{O}_{\text{Auth}}$, which returns true if an input is a valid blockchain transaction. Since Ethereum blocks are self-authenticated using Merkle trees [64, 250], in principle we can realize $\mathcal{O}_{\text{Auth}}$ by including an Ethereum client in the TCB. Doing so drastically increases the code footprint, however, as the core Ethereum implementation is about 50k lines of C++. Similarly, a smart contract could authenticate messages from an SGX by checking attestations, but implementing this verification in a smart contract would be error-prone and computationally (and thus financially) expensive.

Instead we propose two general techniques to avoid these calls and thereby minimize code size in the TCB. The first applies to any hybrid system where one TCB component is a blockchain contract. The second applies to any hybrid system where the TCB components communicate only to make and respond to requests.

Binding \mathcal{T}_{Off} to \mathcal{W}_{TC} . Due to the speed and cost of computation in the on-chain TCB, we wish to avoid implementing signature verification (e.g. Intel’s EPID). There does exist a precompiled Ethereum contract to verify ECDSA signatures [250], but the operation requires a high gas cost. Instead, we describe here how to bind the identity of \mathcal{T}_{Off} to an Ethereum wallet, which allows \mathcal{T}_{On} to simply check

the message sender, which is already verified as part of Ethereum’s transaction protocol.

The key observation is that information can only be inserted into the Ethereum blockchain as a transaction from a wallet. Thus, the only way the Relay can relay messages from \mathcal{T}_{Off} to \mathcal{T}_{On} is through a wallet \mathcal{W}_{TC} . Since Ethereum itself already verifies signatures on transactions (i.e., users interact with Ethereum through an authenticated channel), we can piggyback verification of \mathcal{T}_{Off} signatures on top of the existing transaction signature verification mechanism. Simply put, the \mathcal{T}_{Off} creates \mathcal{W}_{TC} with a fresh public key pk_{Off} whose secret is known only to \mathcal{T}_{Off} .

To make this idea work fully, the public key pk_{Off} must be hardcoded into \mathcal{T}_{On} . A client creating or relying on a contract that uses \mathcal{T}_{On} is responsible for ensuring that this hardcoded pk_{Off} has an appropriate SGX attestation before interacting with \mathcal{T}_{On} . Letting *Verify* denote a verification algorithm for EPID signatures, Figure 2.5 gives the protocol for a client to check that \mathcal{T}_{On} is backed by a valid \mathcal{T}_{Off} instance. (We omit the modeling here of IAS online revocation checks.)

In summary, by assuming that relying clients have verified an attestation of \mathcal{T}_{Off} , we can assume that datagrams sent from \mathcal{W}_{TC} are trusted to originate from \mathcal{T}_{Off} . This eliminates the need to do costly EPID signature verification in \mathcal{T}_{On} .

Additionally, SGX can seal pk_{Off} in non-volatile storage while protecting integrity and confidentiality [150, 32], allowing us to maintain the same binding through server restarts.

User: offline verification of SGX attestation

Inputs: $pk_{sgx}, pk_{off}, \mathcal{T}_{off}, \sigma_{att}$

Verify:

Assert \mathcal{T}_{off} is the expected enclave code

Assert $\Sigma_{sgx}.Verify(pk_{sgx}, \sigma_{att}, (\mathcal{T}_{off}, pk_{off}))$

Assert \mathcal{T}_{on} is correct and parametrized with pk_{off}

// now okay to rely on \mathcal{T}_{on}

Figure 2.5: A client checks an SGX attestation of the enclave’s code \mathcal{T}_{off} and public key pk_{off} . The client also checks that pk_{off} is hardcoded into blockchain contract \mathcal{T}_{on} before using \mathcal{T}_{on} .

Eliminating \mathcal{O}_{Auth} . To eliminate the need to call \mathcal{O}_{Auth} from \mathcal{T}_{off} , we leverage the fact that all messages from \mathcal{T}_{off} to \mathcal{T}_{on} are responses to existing requests. Instead of verifying request parameters in \mathcal{T}_{off} , we can verify in \mathcal{T}_{on} that \mathcal{T}_{off} responded to the correct request. For each request, \mathcal{T}_{on} stores the parameters of that request. In each response, \mathcal{T}_{off} includes the parameters it used to fulfill the request. \mathcal{T}_{on} can then check that the parameters in a response match the stored parameters and, if not, and simply reject. Storing parameters and checking equality are simple operations, so this vastly simpler than calling \mathcal{O}_{Auth} inside \mathcal{T}_{off} .

This approach may appear to open new attacks (e.g., the Relay can send bogus requests to which the \mathcal{T}_{off} respond). As we prove in Section 2.8, however, all such attacks reduce to DoS attacks from the network or the Relay—attacks to which hybrid TCB systems are inherently susceptible and which we do not aim to protect against in TC.

2.6 Town Crier Protocol

We now present some preliminaries followed by the TC protocol. For simplicity, we assume a single instance of $\text{prog}_{\text{encl}}$, although our architecture could scale up to multiple enclaves and even multiple hosts.

To ensure gas sustainability, we require that requesters make gas payments up front as Ether. C_{TC} then reimburses the gas costs of TC. By having a trusted component perform the reimbursement, we are also able to guarantee that a malicious TC cannot steal an honest user's money without delivering valid data.

Notation. We use $\text{msg}.m_i$ to label messages corresponding to those in Fig. 2.2. For payment, let $\$g$ denote gas and $\$f$ to denote non-gas currency. In both cases $\$$ is a type annotation and the letter denotes the numerical amount. For simplicity, we assume that gas and currency adopt the same units (allowing us to avoid explicit conversions). We use the following identifiers to denote currency and gas amounts.

| | | |
|---------------------|--------------------|--|
| $\$f$ | | Currency a requester deposits to refund Town Crier's gas expenditure to deliver a datagram |
| $\$g_{\text{req}}$ | $\$g_{\text{dvr}}$ | GASLIMIT when invoking Request , Deliver , or Cancel , respectively |
| $\$g_{\text{cncl}}$ | | |
| $\$g_{\text{clbk}}$ | | GASLIMIT for callback while executing Deliver , set to the max value that can be reimbursed |
| $\$G_{\text{min}}$ | | Gas required for Deliver excluding callback |
| $\$G_{\text{max}}$ | | Maximum gas TC can provide to invoke Deliver |
| $\$G_{\text{cncl}}$ | | Gas needed to invoke Cancel |
| $\$G_{\emptyset}$ | | Gas needed for Deliver on a canceled request |

$\$G_{\text{min}}$, $\$G_{\text{max}}$, $\$G_{\text{cncl}}$, and $\$G_{\emptyset}$ are system constants, $\$f$ is chosen by the requester (and may be malicious if the requester is dishonest), and $\$g_{\text{dvr}}$ is chosen by the TC Enclave when calling **Deliver**. Though $\$g_{\text{req}}$ and $\$g_{\text{cncl}}$ are set by the requester, a user-initiated transaction will abort if they are too small, so we need not worry about the values.

Initialization. TC deposits at least $\$G_{\text{max}}$ into the \mathcal{W}_{TC} .

The TC Contract \mathcal{C}_{TC} . The TC Contract accepts a datagram request with fee $\$f$ from \mathcal{C}_U , assigns it a unique id, and records it. The Town Crier Relay \mathcal{R} monitors requests and forwards them to the Enclave. As we discussed in Section 2.5.2, upon receipt of a response from \mathcal{W}_{TC} , \mathcal{C}_{TC} verifies that $\text{params}' = \text{params}$ to ensure validity. If the request is valid, \mathcal{C}_{TC} forwards the resulting datagram data by calling the callback specified in the initial request. To ensure that all gas spent

can be reimbursed, \mathcal{C}_{TC} sets $\$g_{clbk} := \$f - \$G_{min}$ for this sub-call. \mathcal{C}_{TC} is specified fully in Fig. 2.6. Here, Call denotes a call to a contact entry point.

| Town Crier blockchain contract \mathcal{C}_{TC} with fees | |
|---|---|
| Initialize: | Counter := 0 |
| Request: | On rcv (params, callback, $\$f$, $\$g_{req}$) from some \mathcal{C}_U : Assert $\$G_{min} \leq \$f \leq \$G_{max}$ id := Counter; Counter := Counter + 1 Store (id, params, callback, $\$f$, \mathcal{C}_U) <i>// msg.m₁</i> <i>// $\\$f$ held by contract</i> |
| Deliver: | On rcv (id, params, data, $\$g_{dvr}$) from \mathcal{W}_{TC} : (1) If isCanceled[id] and not isDelivered[id] Set isDelivered[id] (2) Send $\$G_{\emptyset}$ to \mathcal{W}_{TC} Return Retrieve stored (id, params', callback, $\$f$, $_$) <i>// abort if not found</i> Assert params = params' and $\$f \leq \g_{dvr} and isDelivered[id] not set Set isDelievered[id] (3) Send $\$f$ to \mathcal{W}_{TC} Set $\$g_{clbk} := \$f - \$G_{min}$ (4) Call callback(data) with gas $\$g_{clbk}$ <i>// msg.m₄</i> |
| Cancel: | On rcv (id, $\$g_{cncl}$) from \mathcal{C}_U : Retrieve stored (id, $_$, $_$, $\$f$, \mathcal{C}'_U) <i>// abort if not found</i> Assert $\mathcal{C}_U = \mathcal{C}'_U$ and $\$f \geq \G_{\emptyset} and isDelivered[id] not set and isCanceled[id] not set Set isCanceled[id] (5) Send $(\$f - \$G_{\emptyset})$ to \mathcal{C}_U <i>// hold $\\$G_{\emptyset}$</i> |

Figure 2.6: TC contract \mathcal{C}_{TC} reflecting fees. The last argument of each function is the GASLIMIT provided.

The Relay \mathcal{R} . As noted in Section 2.3, \mathcal{R} bridges the gap between the Enclave and the blockchain in three ways.

1. It scrapes the blockchain and monitors \mathcal{C}_{TC} for new requests (id, params).

2. It boots the Enclave with $\text{prog}_{\text{encl}}.\mathbf{Initialize}()$ and calls $\text{prog}_{\text{encl}}.\mathbf{Resume}(\text{id}, \text{params})$ on incoming requests.
3. It forwards datagrams from the Enclave to the blockchain.

Recall that it forwards already-signed transactions to the blockchain as \mathcal{W}_{TC} . The program for \mathcal{R} is shown in Fig. 2.7. The function `AuthSend` inserts a transaction to blockchain (“as \mathcal{W}_{TC} ” means the transaction is already signed with sk_{TC}). An honest Relay will invoke $\text{prog}_{\text{encl}}.\mathbf{Resume}$ exactly once with the parameters of each valid request and never otherwise.

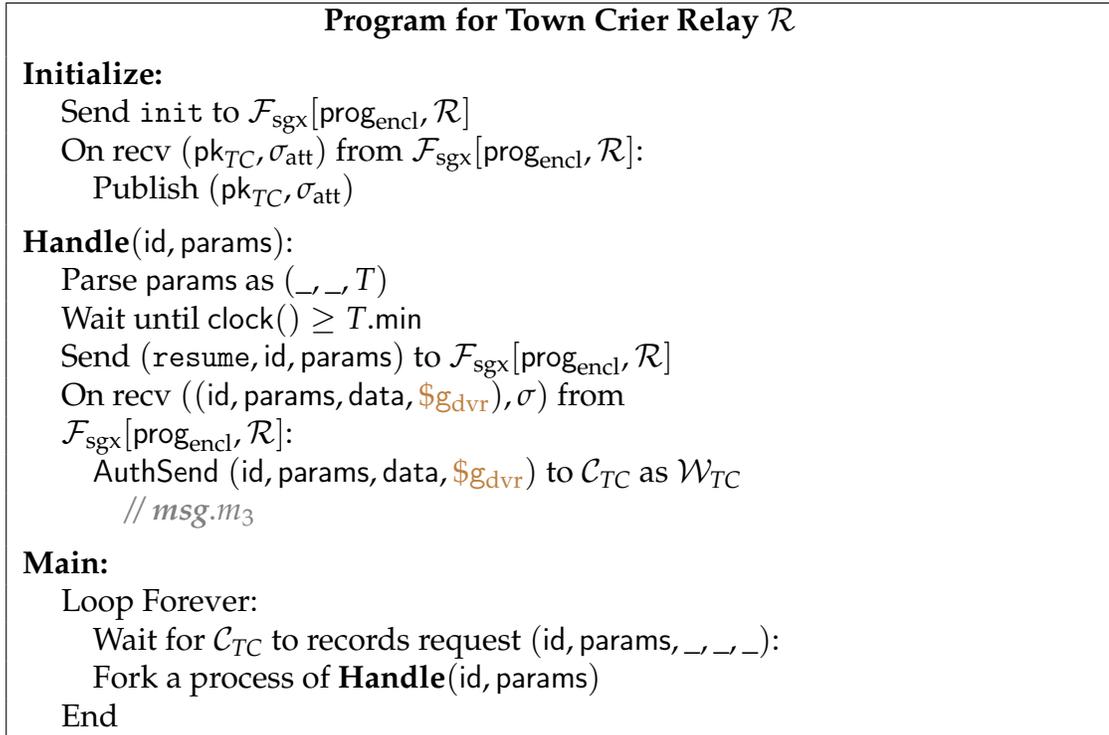


Figure 2.7: The Town Crier Relay \mathcal{R} .

The Enclave $\text{prog}_{\text{encl}}$. When initialized through `Initialize()`, $\text{prog}_{\text{encl}}$ ingests the current wall-clock time; by storing this time and setting a clock reference point, it calibrates its absolute clock. It generates an ECDSA keypair $(\text{pk}_{TC}, \text{sk}_{TC})$

(parameterized as in Ethereum), where pk_{TC} is bound to the $\text{prog}_{\text{encl}}$ instance through insertion into attestations.

Upon a call to **Resume**(id, params), $\text{prog}_{\text{encl}}$ contacts the data source specified by params via HTTPS and checks that the corresponding certificate cert is valid. (We discuss certificate checking in Section 2.7.) Then $\text{prog}_{\text{encl}}$ fetches the requested datagram and returns it to \mathcal{R} along with params , id , and a GASLIMIT $\$g_{\text{dvr}} := \G_{max} , all digitally signed with sk_{TC} . Fig. 2.8 shows the protocol for $\text{prog}_{\text{encl}}$.

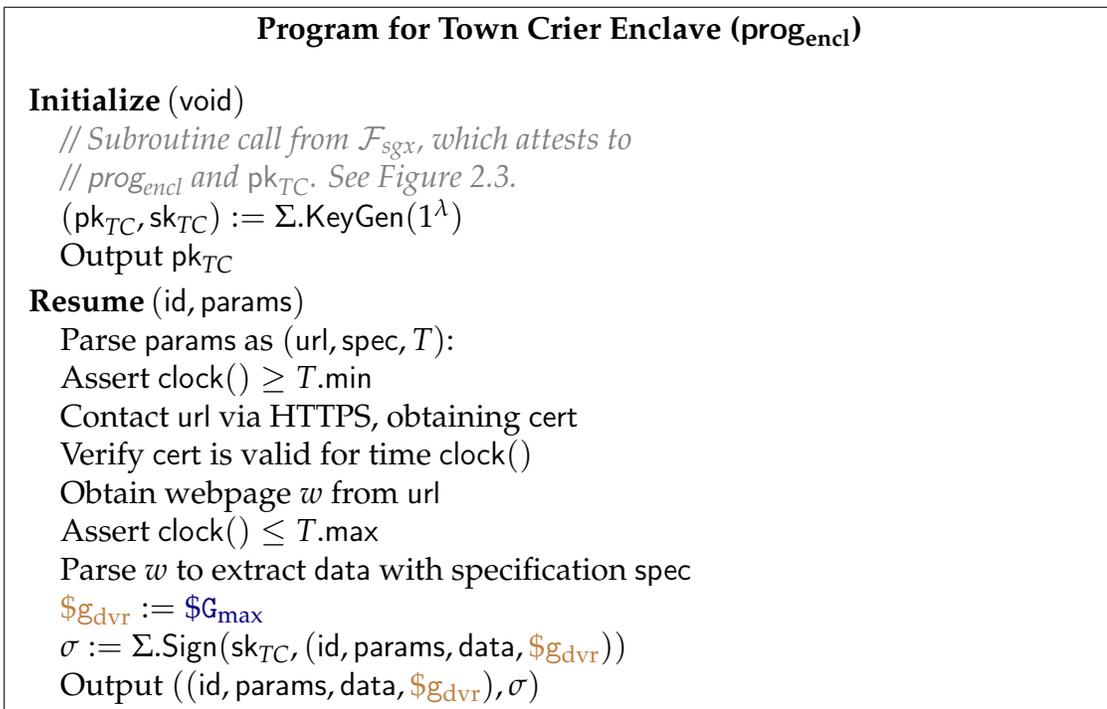


Figure 2.8: The Town Crier Enclave $\text{prog}_{\text{encl}}$.

The Requester Contract \mathcal{C}_U . An honest requester first follows the protocol in Fig. 2.5 to verify the SGX attestation. Then she prepares params and callback, sets $\$g_{\text{req}}$ to the cost of **Request** with params , sets $\$f$ to $\$G_{\text{min}}$ plus the cost of executing callback, and invokes **Request**($\text{params}, \text{callback}, \f) with GASLIMIT $\$g_{\text{req}}$.

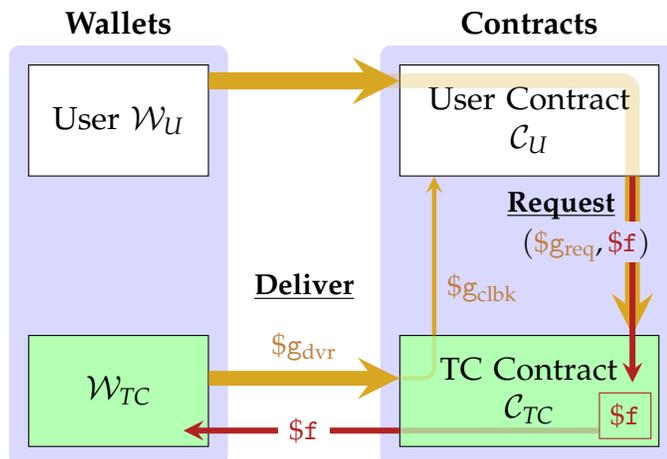


Figure 2.9: Money Flow for a Delivered Request. Red arrows denote flow of money and brown arrows denote gas limits. The thickness of lines indicate the quantity of resources. The $\$g_{clbk}$ arrow is thin because $\$g_{clbk}$ is limited to $\$f - \G_{min} .

If callback is not executed, she can invoke **Cancel**(id) with GASLIMIT $\$G_{cncl}$ to receive a partial refund. An honest requester will invoke **Cancel** at most once for each of her requests and never for any other user's request.

2.6.1 Private and Custom Datagrams

In addition to ordinary datagrams, TC supports *private datagrams*, which are requests where params includes ciphertexts under pk_{TC} . Private datagrams can thus enable confidentiality-preserving applications despite the public readability of the blockchain. *Custom datagrams*, also supported by TC, allow a contract to specify a particular web-scraping target, potentially involving multiple interactions, and thus greatly expand the range of possible relying contracts for TC. We do not treat them in our security proofs, but give examples of both datagram types in Section 2.9.1.

2.6.2 Enhanced Robustness via Replication

Our basic security model for TC assumes the ideal isolation model for SGX described above as well as client trust in data sources. Given various concerns about SGX security [88, 253] and the possible fallibility of data sources, we examine two important ways TC can support hedging. To protect against the compromise of a single SGX instance, contracts may request datagrams from multiple SGX instances and implement majority voting among the responses. This hedge requires increased gas expenditure for additional requests and storage of returned data. Similarly, TC can hedge against the compromise of a data source by scraping multiple sources for the same data and selecting the majority response. We demonstrate both of these mechanisms in our example financial derivative application in Section 2.9.2. (A potential optimization is mentioned in Section 2.11.)

2.7 TC Implementation Details

We now present further, system-level details on the TC contract \mathcal{C}_{TC} and the two parts of the TC server, the Enclave and Relay.

2.7.1 TC Contract

We implement \mathcal{C}_{TC} as described in Section 2.6 in Solidity, a high-level language with JavaScript-like syntax which compiles to Ethereum Virtual Machine bytecode—the language Ethereum contracts use.

In order to handle the most general type of requests—including encrypted parameters—the \mathcal{C}_{TC} implementation requires two parameter fields: an integer specifying the type of request (e.g. flight status) and a byte array of user-specified size. This byte array is parsed and interpreted inside the Enclave, but is treated as an opaque byte array by \mathcal{C}_{TC} . For convenience, we include the timestamp of the current block as an implicit parameter.

To guard against the Relay tampering with request parameters, the \mathcal{C}_{TC} protocol includes `params` as an argument to **Deliver** which validates against stored values. To reduce this cost for large arrays, we store and verify $\text{SHA3-256}(\text{requestType}||\text{timestamp}||\text{paramArray})$. The Relay scrapes the raw values for the Enclave which computes the hash and includes it as an argument to **Deliver**.

As we mentioned in Section 2.4.2, to allow for improved efficiency in client contracts, **Request** returns `id` and **Deliver** includes `id` along with `data` as arguments to `callback`. This allows client contracts to make multiple requests in parallel and differentiate the responses, so it is no longer necessary to create a unique client contract for every request to \mathcal{C}_{TC} .

2.7.2 TC Server

Using the recently released Intel SGX SDK [152], we implemented the TC Server as an SGX-enabled application in C++. In the programming model supported by the SGX SDK, the body of an SGX-enabled application runs as an ordinary user-space application, while a relatively small piece of security-sensitive code runs in the isolated environment of the SGX enclave.

The enclave portion of an SGX-enabled application may be viewed as a shared library exposing an API in the form of *ecalls* [152] to be invoked by the untrusted application. Invocation of an *ecall* transfers control to the enclave; the enclave code runs until it either terminates and explicitly releases control, or some special event (e.g. exception) happens [150]. Again, as we assume SGX provides ideal isolation, the untrusted application cannot observe or alter the execution of *ecalls*.

Enclave programs can make *ocalls* [152] to invoke functions defined outside of the enclave. An *ocall* triggers an exit from the enclave; control is returned once the *ocall* completes. As *ocalls* execute outside the enclave, they must be treated by enclave code as untrusted.

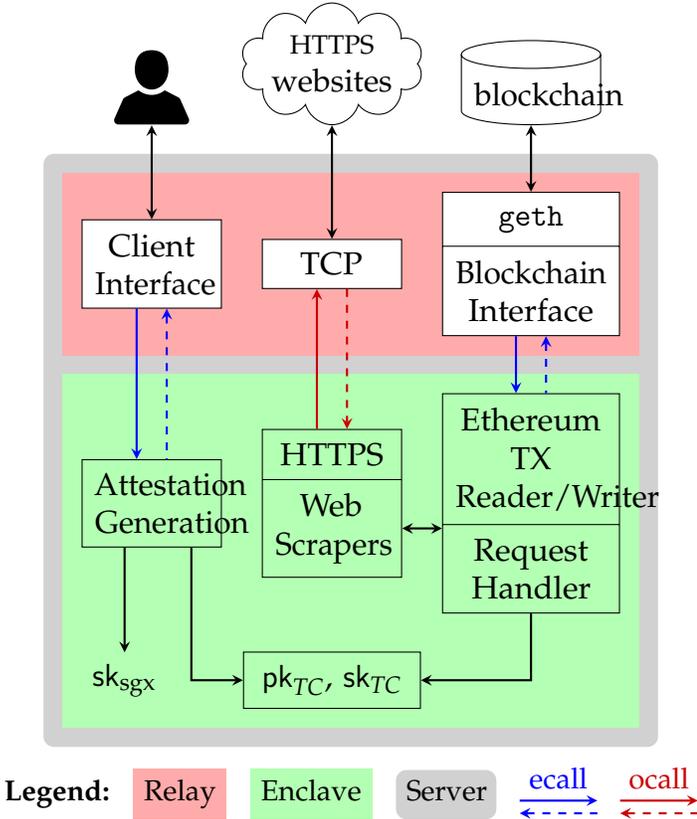


Figure 2.10: Components of TC Server.

For TC, we recall that Fig. 2.8 shows the Enclave code $prog_{\text{encl}}$. Figure 2.7

specifies the operation of the Relay, the untrusted code in TC, which we emphasize again provides essentially only network functionality. We now give details on the services in the Enclave and the Relay and describe their interaction, as summarized in Fig. 2.10.

The Enclave. There are three components to the enclave code `progencl`: an HTTPS service, Web Scrapers, which interact with data sources, and a Request Handler, which services datagram requests.

HTTPS Service. We recall that the enclave does not have direct access to host network functionality. TC thus partitions HTTPS into a trusted layer, consisting of HTTP and TLS code, and an untrusted layer that provides low-layer network service, specifically TCP. This arrangement allows the enclave to establish a secure channel with a web server; the enclave itself performs the TLS handshake with a target server and performs all cryptographic operations internally, while the untrusted process acts as a network interface only. We ported a TLS library (mbedTLS [35]) and HTTP code into the SGX environment. We minimized the HTTP code to meet the web-scraping requirements of TC while keeping the TCB small. To verify certificates presented by remote servers, we hardcoded a collection of root CA certificates into the enclave code; in the first version of TC, the root CAs are identical to those in Chrome. By using its internal, trusted wall-clock time, it is possible to verify that a certificate has not expired. (We briefly discuss revocation in Section 2.11.)

Web Scrapers. We implemented scrapers for our examples in Section 2.9.1 in an ad hoc manner for our initial implementation of TC. We defer more principled, robust approaches to future work.

Request Handler. The Request Handler has two jobs.

1. It ingests a datagram request in Ethereum's serialization format, parses it, and decrypts it (if it is a private-datagram request).
2. It generates an Ethereum transaction containing the requested datagram (and parameter hash), serializes it as a blockchain transaction, signs it using sk_{TC} , and furnishes it to the Relay.

We implemented the Ethereum ABI and RLP which, respectively, specify the serialization of arguments and transactions in Ethereum.

Attestation Generation. Recall in Section 2.2 we mentioned that an *attestation* is an *report* digitally signed by the Intel-provided Quoting Enclave (QE). Therefore two phases are involved in generating att. First, the Enclave calls `sgx_create_report` to generate a report with QE as the target enclave. Then the Relay forwards the report to QE and calls `sgx_get_quote` to get a signed version of the report, namely an attestation.

The Relay. The Relay encompasses three components: A Client Interface, which serves attestations and timestamps, OS services, including networking and time services, and a Blockchain Interface.

Client Interface. As described in Section 2.3, a client starts using TC by requesting and verifying an attestation att and checking the correctness of the clock in the TC enclave using a fresh timestamp. The Client Interface caches att upon initialization of `prog_{encl}`. When it receives a web request from a client for an attestation, it issues an ecall to the enclave to obtain a Unix timestamp signed using sk_{TC} , which it returns to the client along with att. The client verify att using

the Intel Attestation Service (IAS) and then verify the timestamp using pk_{TC} and check it using any trustworthy time service.

OS services. The Enclave relies on the Relay to access networking and wall-clock time (used for initialization) provided by the OS and implemented as `ocalls`.

Blockchain Interface. The Relay's Blockchain Interface monitors the blockchain for incoming requests and places transactions on the blockchain in order to deliver datagrams. The Blockchain Interface incorporates an official Ethereum client, Geth [82]. This Geth client can be configured with a JSON RPC server. The Relay communicates with the blockchain indirectly via RPC calls to this server. For example, to insert a signed transaction, the Relay simply calls `eth_sendRawTransaction` with the byte array of the serialized transaction. We emphasize that, as the enclave holds sk_{TC} , transactions are signed within the enclave.

2.8 Security Analysis

Proofs of theorems in this section appear in Appendix A.2.

Authenticity. Intuitively, authenticity means that an adversary (including a corrupt user, Relay, or collusion thereof) cannot convince \mathcal{C}_{TC} to accept a datagram that differs from the expected content obtained by crawling the specified url at the specified time. In our formal definition, we assume that the user and \mathcal{C}_{TC} behave honestly. Recall that the user must verify upfront the attestation σ_{att}

that vouches for the enclave’s public key pk_{TC} .

Definition 2.2 (Authenticity of Data Feed). *We say that the TC protocol satisfies Authenticity of Data Feed if, for any polynomial-time adversary \mathcal{A} that can interact arbitrarily with \mathcal{F}_{sgx} , \mathcal{A} cannot cause an honest verifier to accept $(\text{pk}_{\text{TC}}, \sigma_{\text{att}}, \text{params} := (\text{url}, \text{pk}_{\text{url}}, T), \text{data}, \sigma)$ where data is not the contents of url with the public key pk_{url} at time T ($\text{prog}_{\text{encl}}.\text{Resume}(\text{id}, \text{params})$ in our model). More formally, for any probabilistic polynomial-time adversary \mathcal{A} ,*

$$\Pr \left[\begin{array}{l} (\text{pk}_{\text{TC}}, \sigma_{\text{att}}, \text{id}, \text{params}, \text{data}, \sigma) \leftarrow (\text{pk}_{\text{TC}}, \sigma_{\text{att}}, \text{id}, \text{params}, \text{data}, \sigma) \leftarrow (\text{pk}_{\text{TC}}, \sigma_{\text{att}}, \text{id}, \text{params}, \text{data}, \sigma) \\ \left(\Sigma_{\text{sgx}}.\text{Verify}(\text{pk}_{\text{sgx}}, \sigma_{\text{att}}, (\text{prog}_{\text{encl}}, \text{pk}_{\text{TC}})) = 1 \right) \wedge \\ \left(\Sigma.\text{Verify}(\text{pk}_{\text{TC}}, \text{id}, \text{params}, \text{data}) = 1 \right) \wedge \\ \text{data} \neq \text{prog}_{\text{encl}}.\text{Resume}(\text{id}, \text{params}) \end{array} \right] \leq \text{negl}(\lambda),$$

for security parameter λ .

Theorem 2.1 (Authenticity). *Assume that Σ_{sgx} and Σ are secure signature schemes. Then, the TC protocol achieves authenticity of data feed under Definition 2.2.³*

Fee Safety. Our protocol in Section 2.6 ensures that an honest Town Crier will not run out of money and that an honest requester will not pay excessive fees.

Theorem 2.2 (Gas Sustainability). *Town Crier is $\$G_{\text{max}}$ -gas sustainable.*

An honest user should only have to pay for computation that is executed honestly on her behalf. If a valid datagram is delivered, this is a constant value plus the cost of executing callback. Otherwise the requester should be able to

³Recall that we model SGX’s group signature as a regular signature scheme under a manufacturer public key pk_{sgx} using the model in [230].

recover the cost of executing **Deliver**. For Theorem 2.2 to hold, C_{TC} must retain a small fee on cancellation, but we allow the user to recover all but this small constant amount. We now formalize this intuition.

Theorem 2.3 (Fair Expenditure for Honest Requester). *For any params and callback, let G_{req} and F be the honestly-chosen values of G_{req} and f , respectively, when submitting the request (params, callback, f , G_{req}). For any such request submitted by an honest user, one of the following holds:*

- *callback is invoked with a valid datagram matching the request parameters params, and the requester spends at most $G_{req} + G_{cncl} + F$;*
- *The requester spends at most $G_{req} + G_{cncl} + G_{\emptyset}$.*

Other security concerns. In Section 2.6.2, we addressed concerns about attacks outside the SGX isolation model embraced in the basic TC protocol. A threat we do not address in TC is the risk of traffic analysis by a network adversary or compromised Relay against confidential applications (e.g., with private datagrams), although we briefly discuss the issue in Section 2.9.1. We also note that while TC assumes the correctness of data sources, if a scraping failure occurs, TC delivers an empty datagram, enabling relying contracts to fail gracefully.

2.9 Experiments

We implemented three showcase applications which we plan to launch together with TC. We provide a brief description of our applications followed by cost and performance measurements. We refer the reader to Appendix A.3 for more details on the applications and code samples.

2.9.1 Requesting Contracts

Financial Derivative (CashSettledPut). Financial derivatives are among the most commonly cited smart contract applications, and exemplify the need for a data feed on financial instruments. We implemented an example contract `CashSettledPut` for a *cash-settled put option*. This is an agreement for one party to buy an asset from the other at an agreed upon price on or before a particular date. It is “cash-settled” in that the sale is implicit, i.e., no asset changes hands, only cash reflecting the asset’s value.

Flight Insurance (FlightIns). Flight insurance indemnifies a purchaser should her flight be delayed or canceled. We have implemented a simple flight insurance contract called `FlightIns`. Our implementation showcases TC’s *private-datagram* feature to address an obvious concern: customers may not wish to reveal their travel plans publicly on the blockchain. Roughly speaking, a customer submits to \mathcal{C}_{TC} a request $\text{Enc}_{pk_{TC}}(\text{req})$ encrypted under Town Crier enclave’s public key pk_{TC} . The enclave decrypts `req` and checks that it is well-formed (e.g., submitted sufficiently long before the flight time). The enclave will then fetch the flight information from a target website at a specified later time, and send to \mathcal{C}_{TC} a datagram indicating whether the flight is delayed or canceled. Finally, to avoid leaking information through timing (e.g., when the flight information website is accessed or datagram sent), random delays are introduced.

Steam Marketplace (SteamTrade). Authenticated data feeds and smart contracts can enable fair exchange of digital goods between Internet users who do not have pre-established trust. We have developed an example application

| | CashSettledPut | | | | | FlightIns | | | | | SteamTrade | | | | |
|---------------|----------------|------|------------|------------|------------|-----------|------|------------|------------|------------|------------|------|------------|------------|------------|
| | mean | % | t_{\max} | t_{\min} | σ_t | mean | % | t_{\max} | t_{\min} | σ_t | mean | % | t_{\max} | t_{\min} | σ_t |
| Ctx. switch | 1.00 | 0.6 | 3.12 | 0.25 | 0.31 | 1.23 | 0.24 | 2.94 | 0.17 | 0.32 | 1.17 | 0.20 | 3.25 | 0.36 | 0.35 |
| Web scraper | 157 | 87.2 | 258 | 135 | 18 | 482 | 95.4 | 600 | 418 | 31 | 576 | 96.2 | 765 | 489 | 52 |
| Sign | 20.2 | 11.2 | 26.6 | 18.7 | 1.52 | 20.5 | 4.0 | 25.3 | 18.9 | 1.4 | 20.3 | 3.4 | 24.8 | 18.8 | 1.28 |
| Serialization | 0.40 | 0.2 | 0.84 | 0.24 | 0.08 | 0.38 | 0.08 | 0.67 | 0.20 | 0.08 | 0.39 | 0.07 | 0.65 | 0.24 | 0.09 |
| Total | 180 | 100 | 284 | 158 | 18 | 505 | 100 | 623 | 439 | 31 | 599 | 100 | 787 | 510 | 52 |

Table 2.1: Enclave response time t , with profiling breakdown. All times are in **milliseconds**. We executed 500 experimental runs, and report the statistics including the average (**mean**), proportion (%), maximum (t_{\max}), minimum (t_{\min}), and standard deviation (σ_t). Note that **Total** is the end-to-end response time as defined in *Enclave Response Time*. Times may not sum to this total due to minor unprofiled overhead.

supporting fair trade of virtual items for Steam [22], an online gaming platform that supports thousands of games and maintains its own marketplace, where users can trade, buy, and sell games and other virtual items. We implemented a contract for the sale of games and items for Ether that showcases TC’s support for *custom datagrams* through the use of Steam’s access-controlled API. In our implementation, the seller sends $\text{Enc}_{\text{pk}_{TC}}(\text{account credentials, req})$ to \mathcal{C}_{TC} , such that the Enclave can log in as the seller and determine from the web-page whether the virtual item has been shipped.

2.9.2 Measurements

We evaluated the performance of TC on a Dell Inspiron 13-7359 laptop with an Intel i7-6500U CPU and 8.00GB memory, one of the few SGX-enabled systems commercially available at the time of writing. We show that on this single host—not even a server, but a consumer device—our implementation of TC can easily process transactions at the peak global rate of Bitcoin, currently the most heavily loaded decentralized blockchain.

We report mean run times (with the standard deviation in parenthesis) over 100 trials.

TCB Size. The trusted computing base (TCB) of Town Crier includes the Enclave and TC Contract. The Enclave consists of approximately 46.4k lines of C/C++ code, the vast majority of which (42.7k lines) is the modified mbedTLS library [35]. The source code of mbedTLS has been widely deployed and tested, while the remainder of the Enclave codebase is small enough to admit formal verification. The TC Contract is also compact; it consists of approximately 120 lines of Solidity code.

Enclave Response Time. We measured the enclave response time for handling a TC request, defined as the interval between (1) the Relay sending a request to the enclave and (2) the Relay receiving a response from the enclave.

Table 2.1 summarizes the total enclave response time as well as its breakdown over 500 runs. For the three applications we implemented, the enclave response time ranges from **180 ms** to **599 ms**. The response time is clearly dominated by the web scraper time, i.e., the time it takes to fetch the requested information from a website. Among the three applications evaluated, SteamTrade has the longest web scraper time, as it interacts with the target website over multiple roundtrips to fetch the desired datagram.

Transaction Throughput. We performed a sequence of experiments measuring the transaction throughput while scaling up the number of concurrently running enclaves on our single SGX-enabled host from 1 to 20. 20 TC enclaves is the

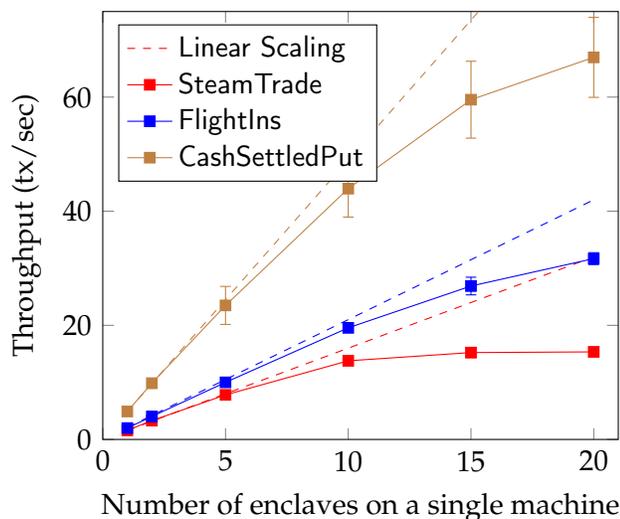


Figure 2.11: Throughput on a single SGX machine. The x-axis is the number of concurrent enclaves and the y-axis is the number of tx/sec. Dashed lines indicate the ideal scaling for each application, and error bars, the standard deviation. We ran 20 rounds of experiments (each round processing 1000 transactions in parallel).

maximum possible given the enclave memory constraints on the specific machine model we used. Figure 2.11 shows that, for the three applications evaluated, a **single SGX machine can handle 15 to 65 tx/sec.**

Several significant data points show how effectively TC can serve the needs of today’s blockchains for authenticated data: Ethereum currently handles under 1 tx/sec on average. Bitcoin today handles slightly more than 3 tx/sec, and its maximum throughput (with full block utilization) is roughly 7 tx/sec. We know of no measurement study of the throughput bound of the Ethereum peer-to-peer network. Recent work [91] indicates that Bitcoin cannot scale beyond 26 tx/sec without a protocol redesign. Thus, with few hosts TC can easily meet the data feed demands of even future decentralized blockchains.

Gas Costs. Currently 1 gas costs 5×10^{-8} Ether, so at the exchange rate of \$15 per Ether, \$1 buys 1.3 million gas. Here we provide costs for our implementation components.

The callback-independent portion of **Deliver** costs about 35,000 gas (2.6¢), so this is the value of $\$G_{\min}$. We set $\$G_{\max} = 3,100,000$ gas (\$2.33), as this is approximately Ethereum’s maximum GASLIMIT. The cost for executing **Request** is approximately 120,000 gas (9¢) of fixed cost, plus 2500 gas (0.19¢) for every 32 bytes of request parameters. The cost to execute **Cancel** is 62500 gas (4.7¢) including the gas cost $\$G_{\text{cncl}}$ and the refund $\$G_{\emptyset}$ paid to TC should **Deliver** be called after **Cancel**.

The total callback-independent cost of acquiring a datagram from TC (i.e., the cost of the datagram, not the application) ranges from 11.9¢ (CashSettledPut) to 12.9¢ (SteamTrade)⁴. The variation results from differing parameter lengths.

Component-Compromise Resilience. For the CashSettledPut application, we implemented and evaluated two modes of majority voting (as in Section 2.6.2):

- 2-out-of-3 majority voting within the enclave, providing robustness against data-source compromise. In our experiments the enclave performed simple sequential scraping of current stock prices from three different data sources: Bloomberg, Google Finance and Yahoo Finance. The enclave response time is roughly 1743 (109) ms in this case (*c.f.* 1058 (88), 423 (34) and 262 (12) ms for each respective data source). There is no change in gas cost, as voting is done inside the SGX enclave. In the future, we will investigate

⁴This cost is for 1 item. Each additional item costs 0.19¢.

parallelization of SGX's thread mechanism, with careful consideration of the security implications.

- 2-out-of-3 majority voting within the requester contract, which provides robustness against SGX compromise. We ran three instances of SGX enclaves, all scraping the same data source. In this scenario the gas cost would increase by a factor of 3 plus an additional 5.85¢. So CashSettledPut would cost 35.6¢ for Deliver without Cancel. The extra 5.85¢ is the cost to store votes until a winner is known.

Offline Measurements. Recall that an enclave requires a one-time setup operation that involves attestation generation. Setting up the TC Enclave takes 49.5 (7.2) ms and attestation generation takes 61.9 (10.7) ms, including 7.65 (0.97) ms for the report, and 54.9 (10.3) ms for the quote.

Recall also that since `clock()` yields only relative time in SGX, TC's absolute clock is calibrated through an externally furnished wall-clock timestamp. A user can verify the correctness of the Enclave absolute clock by requesting a digitally signed timestamp. This procedure is, of course, accurate only to within its end-to-end latency. Our experiments show that the time between Relay transmission of a clock calibration request to the enclave and receipt of a response is 11.4 (1.9) ms of which 10.5 (1.9) ms is to sign the timestamp. To this must be added the wide-area network roundtrip latency, rarely more than a few hundred milliseconds.

2.10 Related Work

Virtual Notary [232, 163] is an early online data attestation service that verifies and digitally signs any of a range of user-requested “factoids” (web page contents, stock prices, etc.) potentially suitable for smart contracts. It predates and does not at present interface with Ethereum.

Several data feeds are deployed today for smart contract systems such as Ethereum. Examples include PriceFeed [18] and Oraclize.it [29]. The latter achieves distributed trust by using a second service called TLSnotary [23], which digitally signs TLS session data. As a result, unlike TC which can flexibly tailor datagrams, Oraclize.it must serve data verbatim from a web session or API call; verbose sources thus mean superfluous data and inflated gas costs. Additionally, these services ultimately rely on the reputations of their (small) providers to ensure data authenticity and cannot support private or custom datagrams. Alternative systems such as SchellingCoin [63] and Augur [5] rely on prediction markets to decentralize trust, creating a heavy reliance on human input and severely constraining their scope and data types.

2.11 Future Work

We plan to develop TC after its initial deployment to incorporate a number of additional features. We discuss a few of those features here.

Freeloading Protection. There are concerns in the Ethereum community about “parasite contracts” that forward or resell datagrams from fee-based data

feeds [241]. As a countermeasure, we plan to deploy the following mechanism in TC inspired by designated verifier proofs [158]. The set of n users $\mathcal{U} = \{U_1, \dots, U_n\}$ of a requesting contract generate an (n, n) -secret-shared key pair $(sk_{\mathcal{U}}, pk_{\mathcal{U}})$. They submit their n individual shares to the TC Enclave (e.g., as ciphertexts under pk_{TC} sent to \mathcal{C}_{TC}).

TC now can sign datagrams using $sk_{\mathcal{U}}$. Each user U_i can be sure individually that a datagram produced by TC is valid, since she did not collude in its creation. Potential parasitic users, however, cannot determine whether the datagram was produced by \mathcal{C}_{TC} or by \mathcal{U} , and thus whether or not it is valid. Such a *source-equivocal datagram* renders parasite contracts less trustworthy and thus less attractive.

Revocation Support. There are two forms of revocation relevant to TC. First, the certificates of data sources may be revoked. Since TC already uses HTTPS, it could easily use the Online Certificate Status Protocol (OCSP) to check TLS certificates. Second, an SGX host could become compromised, prompting revocation of its EPID signatures by Intel. The Intel Attestation Service (IAS) will reportedly disseminate such revocations. Conveniently, clients already use the IAS when checking the attestation σ_{att} , so revocation checking will require no modification to TC.

Hedging Against SGX Compromise. We discussed in Section 2.6.2 how TC can support majority voting across SGX hosts and data sources. Design enhancements to TC could reduce associated latency and gas costs. For SGX voting, we plan to investigate a scheme in which SGX-enabled TC hosts agree on a datagram value X via Byzantine consensus. The hosts may then use a threshold

digital signature scheme to sign the datagram response from \mathcal{W}_{TC} , and each participating host can monitor the blockchain to ensure delivery.

Updating TC's Code. As with any software, we may discover flaws in TC or wish to add new functionality after initial deployment. With TC as described above, however, updating $\text{prog}_{\text{encl}}$ would cause the Enclave to lose access to sk_{TC} and thus be unable to respond to requests in \mathcal{C}_{TC} . The TC operators could set up a new contract \mathcal{C}'_{TC} referencing new keys, but this would be expensive and burdensome for TC's operators and users. While arbitrary code changes would be insecure, we could create a template for user contracts that includes a means to approve upgrades. We plan to investigate this and other mechanisms.

Generalized Custom Datagrams and Within-Enclave Smart-Contract Execution. In our SteamTrade example contract we demonstrated a custom datagram that scrapes a user's online account using her credentials. A more generic approach would allow users to supply their own general-purpose code to TC and data-source-enriched emulation of private contracts as in Hawk [171], but with considerably less computational overhead. Placing such large requests on the blockchain would be prohibitively expensive, but code could easily be loaded into the TC enclave off-chain. Of course, deploying arbitrary user code raises many security and confidentiality concerns which TC would need to address. TC offers a basic framework, however, within which to provide confidential, integrity-protected smart-contract code execution off-chain with trustworthy integration into on-chain smart-contract code.

2.12 Conclusion

In this chapter, we have introduced Town Crier (TC), an authenticated data feed for smart contracts specifically designed to support Ethereum. Use of Intel's new SGX trusted hardware allows TC to serve datagrams with a high degree of trustworthiness. We defined *gas sustainability*, a critical availability property of Ethereum services, and provided techniques for shrinking the size of a hybrid TCB spanning the blockchain and an SGX. We proved in a formal model that TC serves only data from authentic sources, and showed that TC is gas sustainable and minimizes cost to honest users should the code behave maliciously. In experiments involving end-to-end use of the system with the Ethereum blockchain, we demonstrated TC's practicality, cost effectiveness, and flexibility for three example applications. We believe that TC offers a powerful, practical means to address the lack of trustworthy data feeds hampering Ethereum evolution today and that it will support a rich range of applications.

CHAPTER 3

DECO: LIBERATING WEB DATA USING DECENTRALIZED ORACLES FOR TLS

3.1 Introduction

TLS is a powerful, widely deployed protocol that allows users to access web data over confidential, integrity-protected channels. But TLS has a serious limitation: it doesn't allow a user to prove to third parties that a piece of data she has accessed authentically came from a particular website. As a result, data use is often restricted to its point of origin, curtailing data portability by users, a right acknowledged by recent regulations such as GDPR [25].

Specifically, when a user accesses data online via TLS, she cannot securely *export* it, without help (hence permission) from the current data holder. Vast quantities of private data are thus intentionally or unintentionally locked up in the “deep web”—the part of the web that isn't publicly accessible.

To understand the problem, suppose Alice wants to prove to Bob that she's over 18. Currently, age verification services [20] require users to upload IDs and detailed personal information, which raises privacy concerns. But various websites, such as company payroll records or DMV websites, in principle store and serve verified birth dates. Alice could send a screenshot of her birth date from such a site, but this is easily forged. And even if the screenshot could somehow be proven authentic, it would leak information—revealing her exact birth date, not just that she's over 18.

Proposed to prove provenance of online data to smart contracts, *oracles* are a

step towards exporting TLS-protected data to other systems with provenance and integrity assurances. Existing schemes, however, have serious limitations. They either only work with deprecated TLS versions and offer no privacy from the oracle (e.g., TLSNotary [23]) or rely on trusted hardware (e.g., Town Crier in [257] and Chapter 2), against which various attacks have recently emerged, e.g., [61].

Another class of oracle schemes assumes server-side cooperation, mandating that servers install TLS extensions (e.g., [216]) or change application-layer logic (e.g., [75, 255]). Server-facilitated oracle schemes suffer from two fundamental problems. First, they break legacy compatibility, causing a significant barrier to wide adoption. Moreover, such solutions only provide *conditional* exportability because the web servers have the sole discretion to determine which data can be exported, and can censor export attempts at will. A mechanism that allows users to export *any* data they have access to would enable a whole host of currently unrealizable applications.

3.1.1 DECO

To address the above problems, we propose DECO, a decentralized oracle for TLS. Unlike oracle schemes that require per-website support, DECO is source-agnostic and supports *any* website running standard TLS. Unlike solutions that rely on websites' participation, DECO requires no server-side cooperation. Thus a single instance of DECO could enable *anyone* to become an oracle for *any* website.

DECO makes rich Internet data accessible with authenticity and privacy assurances to a wide range of applications, including ones that cannot access the

Internet such as smart contracts. DECO could fundamentally shift today's model of web data dissemination by providing private data delivery with an option for transfer to third parties or public release. This technical capability highlights potential future legal and regulatory challenges, but also anticipates the creation and delivery of appealing new services. Importantly, DECO does not require trusted hardware, unlike alternative approaches that could achieve a similar vision, e.g., [183, 257].

At a high level, the prover commits to a piece of data D and proves to the verifier that D came from a TLS server S and optionally a statement π_D about D . E.g., in the example of proving age, the statement π_D could be the predicate " $D = y/m/d$ is Alice's date of birth and the current date - D is at least 18 years."

Informally, DECO achieves *authenticity*: The verifier is convinced only if the asserted statement about D is true and D is indeed obtained from website S . DECO also provides *privacy* in that the verifier only learns that the statement π_D holds for some D obtained from S .

3.1.2 Technical challenges

Designing DECO with the required security and practical performance, while using legacy-(TLS)-compatible primitives, introduces several important technical challenges. The main challenge stems from the fact that TLS generates symmetric encryption and authentication keys that are *shared* by the client (prover in DECO) and web server. Thus, the client can *forge* arbitrary TLS session data, in the sense of signing the data with valid authentication keys.

To address this challenge, DECO introduces a novel *three-party handshake* protocol among the prover, verifier, and web server that creates an *unforgeable commitment* by the prover to the verifier on a piece of TLS session data D . The verifier can check that D is authentically from the TLS server. From the prover’s perspective, the three-party handshake preserves the security of TLS in presence of a malicious verifier.

Efficient selective opening. After committing to D , the prover proves statements about the commitment. Although arbitrary statements can be supported in theory, we optimize for what are likely to be the most popular applications—revealing only substrings of the response to the verifier. We call such statements *selective opening*. Fine-grained selective opening allows users to hide sensitive information and reduces the input length to the subsequent proofs.

A naïve solution would involve expensive verifiable decryption of TLS records using generic zero-knowledge proofs (ZKPs), but we achieve an orders-of-magnitude efficiency improvement by exploiting the TLS record structure. For example, a direct implementation of verifiable decryption of a TLS record would involve proving correct execution of a circuit of 1024 AES invocations in zero-knowledge, whereas by leveraging the MAC-then-encrypt structure of CBC-HMAC, we achieve the same with only 3 AES invocations.

Context integrity. Selective opening allows the prover to only reveal a substring D' of the server’s response D . However, a substring may mean different things depending on when it appears and a malicious prover could cheat by quoting out of context. Therefore we need to prove not just that D' appears in D , but that it appears in the expected context, i.e., D' has *context integrity* with

respect to D^1 .

Context-integrity attacks can be thwarted if the session content is structured and can be parsed. Fortunately most web data takes this form (e.g., in JSON or HTML). A generic solution is to parse the entire session and prove that the revealed part belongs to the necessary branch of a parse tree. But, under certain constraints that web data generally satisfies, parsing the entire session is not necessary. We propose a novel *two-stage parsing scheme* where the prover pre-processes the session content, and only parses the outcome that is usually much smaller. We draw from the definition of equivalence of programs, as used in programming language theory, to build a formal framework to reason about the security of two-stage parsing schemes. We provide several practical realizations for specific grammars. Our definitions and constructions generalize to other oracles too. For example, it could prevent a generic version of the content-hidden attack mentioned in [216].

3.1.3 Implementation and evaluation

We designed and implemented DECO as a complete end-to-end system. To demonstrate the system’s power, we implemented three applications: 1) a confidentiality-preserving *financial instrument* using smart contracts; 2) converting legacy credentials to *anonymous credentials*; and 3) verifiable claims against *price discrimination*.

Our experiments with these applications show that DECO is highly efficient. For example, for TLS 1.2 in the WAN setting, online time is 2.85s to perform

¹Note that this differs from “contextual integrity” in privacy theory [193].

the three-party handshake and 2.52s for 2PC query execution. It takes 3-10s to generate zero-knowledge proofs for the applications described above. More details are in Section 3.7.

Contributions. In summary, our contributions are as follows:

- We introduce DECO, a provably secure decentralized oracle scheme, along with an implementation and performance evaluation. DECO is the first oracle scheme for modern TLS versions (both 1.2 and 1.3) that doesn't require trusted hardware or server-side modifications. We provide an overview of the protocol in Section 3.3 and specify the full protocol in Section 3.4.
- **Selective opening:** In Section 3.5.1, we introduce a broad class of statements for TLS records that can be proven efficiently in zero-knowledge. They allow users to open only substrings of a session-data commitment. The optimizations achieve substantial efficiency improvement over generic ZKPs.
- **Context-integrity attacks and mitigation:** We identify a new class of context-integrity attacks universal to privacy-preserving oracles (e.g. [216]). In Section 3.5.2, we introduce our mitigation involving a novel, efficient two-stage parsing scheme, along with a formal security analysis, and several practical realizations.
- **Security definitions and proofs:** Oracles are a key part of the smart contract ecosystem, but a coherent security definition has been lacking. We formalize and strengthen existing oracle schemes and present a formal security definition using an ideal functionality in Section 3.3.2. We prove the functionality is securely realized by our protocols in Appendix B.2.

- **Applications and evaluation:** In Section 3.6, we present three representative applications that showcase DECO’s capabilities, and evaluate them in Section 3.7.
- **Legal and compliance considerations:** DECO can export data from websites without their explicit approval or even awareness. We discuss the resulting legal and compliance issues in Section 3.8.

3.2 Background

3.2.1 Transport Layer Security (TLS)

We now provide necessary background on the TLS handshake and record protocols on which DECO builds.

TLS is a family of protocols that provides privacy and data integrity between two communicating applications. Roughly speaking, it consists of two protocols: a handshake protocol that sets up the session using asymmetric cryptography, establishing shared client and server keys for the next protocol, the record protocol, in which data is transmitted with confidentiality and integrity protection using symmetric cryptography.

Handshake. In the handshake protocol, the server and client first agree on a set of cryptographic algorithms (also known as a cipher suite). They then authenticate each other (client authentication optional), and finally securely compute a shared secret to be used for the subsequent record protocol.

DECO supports the recommended elliptic curve DH key exchange with ephemeral secrets (ECDHE [55]).

Record protocol. To transmit application-layer data (e.g., HTTP messages) in TLS, the record protocol first fragments the application data D into fixed sized plaintext *records* $D = (D_1, \dots, D_n)$. Each record is usually padded to a multiple of blocks (e.g., 128 bits). The record protocol then optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level protocols. The specific cryptographic operations depend on the negotiated ciphersuite. DECO supports the AES cipher in two commonly used modes: CBC-HMAC and GCM. We refer readers to [215] for how these primitives are used in TLS.

Differences between TLS 1.2 and 1.3. Throughout the paper we focus on TLS 1.2 and discuss how to generalize our techniques to TLS 1.3 in Section 3.4.1.3. Here we briefly note the major differences between these two TLS versions. TLS 1.3 removes the support for legacy non-AEAD ciphers. The handshake flow has also been restructured. All handshake messages after the ServerHello are now encrypted. Finally, a different key derivation function is used. For a complete description, see [214].

3.2.2 Multi-party computation

Consider a group of n parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, each of whom holds some secret s_i . Secure multi-party computation (MPC) allows them to jointly compute $f(s_1, \dots, s_n)$

without leaking any information other than the output of f , i.e., \mathcal{P}_i learns nothing about $s_{j \neq i}$. Security for MPC protocols generally considers an adversary that corrupts t players and attempts to learn the private information of an honest player. Two-party computation (2PC) refers to the special case of $n = 2$ and $t = 1$. We refer the reader to [177] for a full discussion of the model and formal security definitions.

There are two general approaches to 2PC protocols. Garbled-circuit protocols based on Yao [254] encode f as a boolean circuit, an approach best-suited for bitwise operations (e.g., SHA-256). Other protocols leverage *threshold secret sharing* and are best suited for arithmetic operations. The functions we compute in this paper using 2PC, though, include both bitwise and arithmetic operations. We separate them into two components, and use the optimized garbled-circuit protocol from [247] for the bitwise operations and the secret-sharing based MtA protocol from [128] for the arithmetic operations.

3.3 Overview

In this section we state the problem we try to solve with DECO and present a high-level overview of its architecture.

3.3.1 Problem statement: Decentralized oracles

Broadly, we investigate protocols for building “oracles,” i.e., entities that can prove provenance and properties of online data. The goal is to allow a prover \mathcal{P} to prove to a verifier \mathcal{V} that a piece of data came from a particular website \mathcal{S}

and optionally prove statements about such data in zero-knowledge, keeping the data itself secret. Accessing the data may require private input (e.g., a password) from \mathcal{P} and such private information should be kept secret from \mathcal{V} as well.

We focus on servers running TLS, the most widely deployed security protocol suite on the Internet. However, TLS alone does not prove data provenance. Although TLS uses public-key signatures for authentication, it uses symmetric-key primitives to protect the integrity and confidentiality of exchanged messages, using a shared session key established at the beginning of each session. Hence \mathcal{P} , who knows this symmetric key, cannot prove statements about cryptographically authenticated TLS data to a third party.

A web server itself could assume the role of an oracle, e.g., by simply signing data. However, server-facilitated oracles would not only incur a high adoption cost, but also put users at a disadvantage: the web server could impose arbitrary constraints on the oracle capability. We are interested in a scheme where anyone can prove provenance of any data she can access, without needing to rely on a single, central point of control, such as the web server providing the data.

We tackle these challenges by introducing *decentralized oracles* that don't rely on trusted hardware or cooperation from web servers. The problem is much more challenging than for previous oracles, as it precludes solutions that require servers to modify their code or deploy new software, e.g., [216], or use of prediction markets, e.g., [30, 207], while at the same time going beyond these previous approaches by supporting proofs on arbitrary predicates over data. Another approach, introduced in [257], is to use trusted execution environments (TEEs) such as Intel SGX. The downside is that recent attacks [61] may deter some users from trusting TEEs.

Authenticated data feeds for smart contracts. An important application of oracle protocols is to construct authenticated data feeds (ADFs, as coined in [257]), i.e., data with verifiable provenance and correctness, for smart contracts. Protocols such as [257] generate ADFs by signing TLS data using a key kept secret in a TEE. However, the security of this approach relies on that of TEEs. Using multiple TEEs could help achieve stronger integrity, but not privacy. If a single TEE is broken, TLS session content, including user credentials, can leak from the broken TEE.

DECO operates in a different model. Since smart contracts can't participate in 2PC protocols, they must rely on oracle nodes to participate as \mathcal{V} on their behalf. Therefore we envision DECO being deployed in a decentralized oracle network similar to [113], where a set of independently operated oracles are available for smart contracts to use. Note that oracles running DECO are trusted only for integrity, not for privacy. Smart contracts can further hedge against integrity failures by querying multiple oracles and requiring, e.g., majority agreement, as already supported in [113]. We emphasize that DECO's privacy is preserved even all oracles are compromised. Thus DECO enables users to provide ADFs derived from private data to smart contracts while hiding private data from oracles.

3.3.2 Notation and definitions

We use \mathcal{P} to denote the prover, \mathcal{V} the verifier and \mathcal{S} the TLS server. We use letters in boldface (e.g., \mathbf{M}) to denote vectors and M_i to denote the i th element in \mathbf{M} .

We model the essential properties of an oracle using an ideal functionality $\mathcal{F}_{\text{Oracle}}$ in Fig. 3.1. To separate parallel runs of $\mathcal{F}_{\text{Oracle}}$, all messages are tagged

Functionality $\mathcal{F}_{\text{Oracle}}$ between \mathcal{S}, \mathcal{P} and \mathcal{V}

Input: The prover \mathcal{P} holds some private input θ_s . The verifier \mathcal{V} holds a query template Query and a statement Stmt.

Functionality:

- If at any point during the session, a message $(\text{sid}, \text{receiver}, m)$ with $\text{receiver} \in \{\mathcal{S}, \mathcal{P}, \mathcal{V}\}$ is received from \mathcal{A} , forward (sid, m) to receiver and forward any responses to \mathcal{A} .
- Upon receiving input $(\text{sid}, \text{Query}, \text{Stmt})$ from \mathcal{V} , send $(\text{sid}, \text{Query}, \text{Stmt})$ to \mathcal{P} . Wait for \mathcal{P} to reply with “ok” and θ_s .
- Compute $Q = \text{Query}(\theta_s)$ and send (sid, Q) to \mathcal{S} and record its response (sid, R) . Send $(\text{sid}, |Q|, |R|)$ to \mathcal{A} .
- Send (sid, Q, R) to \mathcal{P} and $(\text{sid}, \text{Stmt}(R), \mathcal{S})$ to \mathcal{V} .

Figure 3.1: The oracle functionality.

with a unique *session id* denoted sid . We refer readers to [70] for details of ideal protocol execution.

$\mathcal{F}_{\text{Oracle}}$ accepts a secret parameter θ_s (e.g., a password) from \mathcal{P} , a query template Query and a statement Stmt from \mathcal{V} . A query template is a function that takes \mathcal{P} 's secret θ_s and returns a complete query, which contains public parameters specified by \mathcal{V} . An example query template would be $\text{Query}(\theta_s) = \text{“stock price of GOOG on Jan 1st, 2020 with API key} = \theta_s\text{”}$. The prover \mathcal{P} can later prove that the query sent to the server is well-formed, i.e., built from the template, without revealing the secret. The statement Stmt is a function that \mathcal{V} wishes to evaluate on the server's response. Following the previous example, as the response R is a number, the following statement would compare it with a threshold: $\text{Stmt}(R) = \text{“}R > \$1,000\text{”}$.

After \mathcal{P} acknowledges the query template and the statement (by sending “ok” and θ_s), $\mathcal{F}_{\text{Oracle}}$ retrieves a response R from \mathcal{S} using a query built from the template. We assume an honest server, so R is the ground truth. $\mathcal{F}_{\text{Oracle}}$ sends $\text{Stmt}(R)$ and the data source to \mathcal{V} .

As stated in Definition 3.1, we are interested in decentralized oracles that don't require any server-side modifications or cooperation, i.e., \mathcal{S} follows the unmodified TLS protocol.

Definition 3.1. *A decentralized oracle protocol for TLS is a three-party protocol $\text{Prot} = (\text{Prot}_{\mathcal{S}}, \text{Prot}_{\mathcal{P}}, \text{Prot}_{\mathcal{V}})$ such that 1) Prot realizes $\mathcal{F}_{\text{Oracle}}$ and 2) $\text{Prot}_{\mathcal{S}}$ is the standard TLS, possibly along with an application-layer protocol.*

Adversarial model and security properties. We consider a static, malicious network adversary \mathcal{A} . Corrupted parties may deviate arbitrarily from the protocol and reveal their states to \mathcal{A} . As a network adversary, \mathcal{A} learns the message length from $\mathcal{F}_{\text{Oracle}}$ since TLS is not length-hiding. We assume \mathcal{P} and \mathcal{V} choose and agree on an appropriate query (e.g., it should be idempotent for most applications) and statement according to the application-layer protocol run by \mathcal{S} .

For a given query Q , denote the server's honest response by $\mathcal{S}(Q)$. We require that security holds when either \mathcal{P} or \mathcal{V} is corrupted. The functionality $\mathcal{F}_{\text{Oracle}}$ reflects the following security guarantees:

- *Prover-integrity:* A malicious \mathcal{P} cannot forge content provenance, nor can she cause \mathcal{S} to accept invalid queries or respond incorrectly to valid ones. Specifically, if the verifier inputs $(\text{Query}, \text{Stmt})$ and outputs (b, \mathcal{S}) , then \mathcal{P} must have sent $Q = \text{Query}(\theta_s)$ to \mathcal{S} in a TLS session, receiving response $R = \mathcal{S}(Q)$ such that $b = \text{Stmt}(R)$.
- *Verifier-integrity:* A malicious \mathcal{V} cannot cause \mathcal{P} to receive incorrect responses. Specifically, if \mathcal{P} outputs (Q, R) then R must be the server's response to query Q submitted by \mathcal{P} , i.e., $R = \mathcal{S}(Q)$.

- *Privacy*: A malicious \mathcal{V} learns only public information (Query, \mathcal{S}) and the evaluation of $\text{Stmt}(R)$.

3.3.3 A strawman protocol

We focus on two widely used representative TLS cipher suites: CBC-HMAC and AES-GCM. Our technique generalizes to other ciphers (e.g., Chacha20-Poly1305, etc.) as well. Throughout this section we use CBC-HMAC to illustrate the ideas, with discussion of GCM deferred to later sections.

TLS uses separate keys for each direction of communication. Unless explicitly specified, we don't distinguish between the two and use k^{Enc} and k^{MAC} to denote session keys for both directions.

In presenting our design of DECO, we start with a strawman protocol and incrementally build up to the full protocol.

A strawman protocol. A strawman protocol that realizes $\mathcal{F}_{\text{Oracle}}$ between $(\mathcal{P}, \mathcal{V})$ is as follows. \mathcal{P} queries the server \mathcal{S} and records all messages sent to and received from the server in $\hat{\mathbf{Q}} = (\hat{Q}_1, \dots, \hat{Q}_n)$ and $\hat{\mathbf{R}} = (\hat{R}_1, \dots, \hat{R}_n)$, respectively. Let $\hat{\mathbf{M}} = (\hat{\mathbf{Q}}, \hat{\mathbf{R}})$ and $(k^{\text{MAC}}, k^{\text{Enc}})$ be the session keys.

She then proves in zero-knowledge that 1) each \hat{R}_i decrypts to $R_i \parallel \sigma_i$, a plain-text record and a MAC tag; 2) each MAC tag σ_i for R_i verifies against k^{MAC} ; and 3) the desired statement evaluates to b on the response, i.e., $b = \text{Stmt}(R)$. Using

the now standard notation introduced in [68], \mathcal{P} computes

$$p_r = \text{ZK-PoK}\{k^{\text{Enc}}, \mathbf{R} : \forall i \in [n], \text{Dec}(k^{\text{Enc}}, \hat{R}_i) = R_i \parallel \sigma_i \\ \wedge \text{Verify}(k^{\text{MAC}}, \sigma_i, R_i) = 1 \wedge \text{Stmt}(\mathbf{R}) = b\}.$$

She also proves that \mathbf{Q} is well-formed as $\mathbf{Q} = \text{Query}(\theta_s)$ similarly in a proof p_q and sends $(p_q, p_r, k^{\text{MAC}}, \hat{\mathbf{M}}, b)$ to \mathcal{V} .

Given that $\hat{\mathbf{M}}$ is an authentic transcript of the TLS session, the prover-integrity property seems to hold. Intuitively, CBC-HMAC ciphertexts bind to the underlying plaintexts, thus $\hat{\mathbf{M}}$ can be treated as secure commitments [139] to the session data. That is, a given $\hat{\mathbf{M}}$ can only be opened (i.e., decrypted and MAC checked) to a unique message. The binding property prevents \mathcal{P} from opening $\hat{\mathbf{M}}$ to a different message other than the original session with the server.

Unfortunately, this intuition is flawed. The strawman protocol fails completely because it *cannot* ensure the authenticity of $\hat{\mathbf{M}}$. The prover \mathcal{P} has the session keys, and thus she can include the encryption of arbitrary messages in $\hat{\mathbf{M}}$.

Moreover, the zero-knowledge proofs that \mathcal{P} needs to construct involve decrypting and hashing the entire transcript, which can be prohibitively expensive. For the protocol to be practical, we need to significantly reduce the cost.

3.3.4 Overview of DECO

The critical failing of our strawman approach is that \mathcal{P} learns the session key before she commits to the session. One key idea in DECO is to withhold the MAC

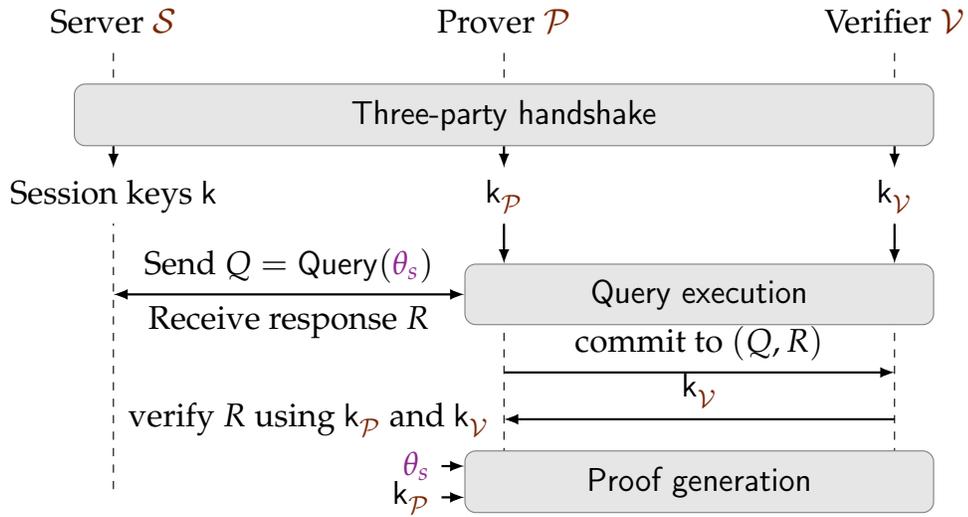


Figure 3.2: An overview of the workflow in DECO. The protocol has three phases: a **three-party handshake** phase to establish session keys in a special format to achieve unforgeability, a **query execution** phase where \mathcal{P} queries the server for data using a query built from the template with her private parameters θ_s , and finally a **proof generation** phase in which \mathcal{P} proves that the query well-formed and the response satisfies the desired condition.

key from \mathcal{P} until *after* she commits. The TLS session between \mathcal{P} and the server \mathcal{S} must still provide confidentiality and integrity. Moreover, the protocol must not degrade performance below the requirements of TLS (e.g., triggering a timeout).

As shown in Fig. 3.2, DECO is a three-phase protocol. The first phase is a novel **three-party handshake** protocol in which the prover \mathcal{P} , the verifier \mathcal{V} , and the TLS server \mathcal{S} establish session keys that are *secret-shared between \mathcal{P} and \mathcal{V}* . After the handshake is a **query execution** phase during which \mathcal{P} accesses the server following the standard TLS protocol, but with help from \mathcal{V} . After \mathcal{P} commits to the query and response, \mathcal{V} reveals her key share. Finally, \mathcal{P} proves statements about the response in a **proof generation** phase.

3.3.4.1 Three-party handshake

Essentially, \mathcal{P} and \mathcal{V} *jointly* act as a TLS client. They negotiate a shared session key with \mathcal{S} in a secret-shared form. We emphasize that this phase, like the rest of DECO, is completely transparent to \mathcal{S} , requiring no server-side modifications.

For the CBC-HMAC cipher suite, at the end of the three-party handshake, \mathcal{P} and \mathcal{V} receive $k_{\mathcal{P}}^{\text{MAC}}$ and $k_{\mathcal{V}}^{\text{MAC}}$ respectively, while \mathcal{S} receives $k^{\text{MAC}} = k_{\mathcal{P}}^{\text{MAC}} + k_{\mathcal{V}}^{\text{MAC}}$. As with the standard handshake, both \mathcal{P} and \mathcal{S} get the encryption key k^{Enc} .

Three-party handshake can make the aforementioned session-data commitment unforgeable as follows. At the end of the session, \mathcal{P} first commits to the session in \hat{M} as before, then \mathcal{V} reveals her share $k_{\mathcal{V}}^{\text{MAC}}$. From \mathcal{V} 's perspective, the three-party handshake protocol ensures that a fresh MAC key (for each direction) is used for every session, despite the influence of a potential malicious prover, and that the keys are unknown to \mathcal{P} until she commits. Without knowledge of the MAC key, \mathcal{P} cannot forge or tamper with session data before committing to it. The unforgeability of the session-data commitment in DECO thus reduces to the unforgeability of the MAC scheme used in TLS.

Other ciphersuites such as GCM can be supported similarly. In GCM, a single key (for each direction) is used for both encryption and MAC. The handshake protocol similarly secret-shares the key between \mathcal{P} and \mathcal{V} . The handshake protocol are presented in Section 3.4.1.

3.3.4.2 Query execution

Since the session keys are secret-shared, as noted, \mathcal{P} and \mathcal{V} execute an interactive protocol to construct a TLS message encrypting the query. \mathcal{P} then sends the message to \mathcal{S} as a standard TLS client. For CBC-HMAC, they compute the MAC tag of the query, while for GCM they perform authenticated encryption. Note that the query is private to \mathcal{P} and should not be leaked to \mathcal{V} . Generic 2PC would be expensive for large queries, so we instead introduce custom 2PC protocols that are orders-of-magnitude more efficient than generic solutions, as presented in Section 3.4.2.

As explained previously, \mathcal{P} commits to the session data \hat{M} before receiving \mathcal{V} 's key share, making the commitment unforgeable. Then \mathcal{P} can verify the integrity of the response, and prove statements about it, which we present now.

3.3.4.3 Proof generation

With unforgeable commitments, if \mathcal{P} opens the commitment \hat{M} completely (i.e., reveals the encryption key) then \mathcal{V} could easily verify the authenticity of \hat{M} by checking MACs on the decryption.

Revealing the encryption key for \hat{M} , however, would breach privacy: it would reveal *all* session data exchanged between \mathcal{P} and \mathcal{S} . In theory, \mathcal{P} could instead prove any statement Stmt over \hat{M} in zero knowledge (i.e., without revealing the encryption key). Generic zero-knowledge proof techniques, though, would be prohibitively expensive for many natural choices of Stmt.

DECO instead introduces two techniques to support efficient proofs for a

broad, general class of statement, namely *selective opening* of a TLS session transcript. Selective opening involves either *revealing* a substring to \mathcal{V} or *redacting*, i.e., excising, a substring, concealing it from \mathcal{V} .

As an example, Fig. 3.3 shows a simplified JSON bank statement for Bob. Suppose Bob (\mathcal{P}) wants to reveal his checking account balance to \mathcal{V} . Revealing the decryption key for his TLS session would be undesirable: it would *also* reveal the entire statement, including his transactions. Instead, using techniques we introduce, Bob can efficiently reveal only the substring in lines 5-7. Alternatively, if he doesn't mind revealing his savings account balance, he might redact his transactions after line 7.

The two selective opening modes, revealing and redacting substrings, are useful privacy protection mechanisms. They can also serve as pre-processing for a subsequent zero-knowledge proof. For example, Bob might wish to prove that he has an account with a balance larger than \$1000, without revealing the actual balance. He would then prove in zero knowledge a predicate ("balance > \$1000") over the substring that includes his checking account balance.

Selective opening *alone*, however, is not enough for many applications. This is because the *context* of a substring affects its meaning. Without what we call *context integrity*, \mathcal{P} could cheat and reveal a substring that falsely appears to prove a claim to \mathcal{V} . For example, Bob might not have a balance above \$1000. After viewing his bank statement, though, he might in the same TLS session post a message to customer service with the substring "balance": \$5000 and then view his pending messages (in a form of reflection attack). He could then reveal this substring to fool \mathcal{V} .

```

1   {"name": "Bob",
2   "savings a/c": {
3     "balance": $5000
4   },
5   "checking a/c": {
6     "balance": $2000
7   },
8   "transactions": {...}}

```

Figure 3.3: Example bank statement to demonstrate selective opening and context-integrity attacks.

Various sanitization heuristics on prover-supplied inputs to \mathcal{V} , e.g., truncating session transcripts, could potentially prevent some such attacks, but, like other forms of web application input sanitization, are fragile and prone to attack [225].

Instead, we introduce a rigorous technique by which session data are explicitly but confidentially parsed. We call this technique *zero-knowledge two-stage parsing*. The idea is that \mathcal{P} parses \hat{M} locally in a first stage and then proves to \mathcal{V} a statement in zero knowledge about constraints on a resulting substring. For example, in our banking example, if bank-supplied key-value stores are always escaped with a distinguished character λ , then Bob could prove a correct balance by extracting via local parsing and revealing to \mathcal{V} a substring "balance": \$5000 preceded by λ . We show for a very common class of web API grammars (unique keys) that this two-phase approach yields much more efficient proofs than more generic techniques.

Section 3.5 gives more details on proof generation in DECO.

3.4 The DECO protocol

We now specify the full DECO protocol, which consists of a three-party handshake in Section 3.4.1, followed by 2PC protocols for query execution in Section 3.4.2, and a proof generation phase. We prove its security in Section 3.4.4.

3.4.1 Three-party handshake

The goal of the three-party handshake (3P-HS) is to secret-share between the prover \mathcal{P} and verifier \mathcal{V} the session keys used in a TLS session with server \mathcal{S} , in a way that is completely transparent to \mathcal{S} . We first focus on CBC-HMAC for exposition, then adapt the protocol to support GCM.

As with the standard TLS handshake, 3P-HS is two-step: first, \mathcal{P} and \mathcal{V} compute additive shares of a secret value $Z \in EC(\mathbb{F}_p)$ shared with the server through a TLS-compatible key exchange protocol. ECDHE is the recommended and the focus here; second, \mathcal{P} and \mathcal{V} derive secret-shared session keys by securely evaluating the TLS-PRF [215] with their shares of Z as inputs.

The full protocol is specified in Fig. 3.4. Below we specify the protocol for each step. We give text descriptions so formal specifications are not required for understanding.

3.4.1.1 Step 1: key exchange

Let $EC(\mathbb{F}_p)$ denote the EC group used in ECDHE and G its generator.

The protocol of three-party handshake (3P-HS) protocol among \mathcal{P} , \mathcal{V} and \mathcal{S}

Public information: Let EC be the Elliptic Curve used in ECDHE over \mathbb{F}_p with order p , G a parameter, and Y_S the server public key.

Output: \mathcal{P} and \mathcal{V} output k_P^{MAC} and k_V^{MAC} respectively, while the TLS server outputs $k^{\text{MAC}} = k_P^{\text{MAC}} + k_V^{\text{MAC}}$. Besides, both \mathcal{S} and \mathcal{P} outputs k^{Enc} .

TLS server \mathcal{S} : follow the standard TLS protocol.

Prover \mathcal{P} :

On initialization: \mathcal{P} samples $r_c \leftarrow_{\$} \{0, 1\}^{256}$ and sends $\text{ClientHello}(r_c)$ to \mathcal{S} to start a standard TLS handshake.

On receiving $\text{ServerHello}(r_s)$, $\text{ServerKeyEx}(Y, \sigma, \text{cert})$ from \mathcal{S} :

- \mathcal{P} verifies that cert is a valid certificate and that σ is a valid signature over (r_c, r_s, Y_S) signed by a key contained in cert . \mathcal{P} sends $(r_c, r_s, Y_S, \sigma, \text{cert})$ to \mathcal{V} .
- \mathcal{V} checks cert and σ similarly. \mathcal{V} then samples $s_V \leftarrow_{\$} \mathbb{F}_p$ and computes $Y_V = s_V \cdot G$. Send Y_V to \mathcal{P} .
- \mathcal{P} samples $s_P \leftarrow_{\$} \mathbb{F}_p$ and computes $Y_P = s_P \cdot G$. Send $\text{ClientKeyEx}(Y_P + Y_V)$ to \mathcal{S} .
- \mathcal{P} and \mathcal{V} run ECtF to compute a sharing of the x -coordinate of $Y_P + Y_V$, denoted z_P, z_V .
- \mathcal{P} (and \mathcal{V}) send z_P (and z_V) to $\mathcal{F}_{2\text{PC}}^{\text{hs}}$ (specified below) to compute shares of session keys and the master secret. \mathcal{P} receives $(k^{\text{Enc}}, k_P^{\text{MAC}}, m_P)$, while \mathcal{V} receives (k_V^{MAC}, m_V) .
- \mathcal{P} computes a hash (denoted h) of the handshake messages sent and received thus far, and runs 2PC-PRF with \mathcal{V} to compute $s = \text{PRF}(m_P \oplus m_V, \text{"client finished"}, h)$ on the hash of the handshake messages and send a $\text{Finished}(s)$ to \mathcal{S} .

On receiving other messages from \mathcal{S} :

- If it's $\text{Finished}(s)$, \mathcal{P} and \mathcal{V} run a 2PC to check $s \stackrel{?}{=} \text{PRF}(m_P \oplus m_V, \text{"server finished"}, h)$ and abort if not.
- Otherwise respond according to the standard TLS protocol.

$\mathcal{F}_{2\text{PC}}^{\text{hs}}$ with \mathcal{P} and \mathcal{V}

Public Input: nonce r_c, r_s

Private Input: $z_P \in \mathbb{F}_p$ from \mathcal{P} ; $z_V \in \mathbb{F}_p^2$ from \mathcal{V}

- $z := z_P + z_V$
- $m := \text{PRF}(z, \text{"master secret"}, r_c \| r_s)$ (truncate at 48 bytes)
- $k^{\text{MAC}}, k^{\text{Enc}} := \text{PRF}(m, \text{"key expansion"}, r_s \| r_c)$ // key expansion
- Sample $r_k, r_m \leftarrow_{\$} \mathbb{F}_p$. Send $(k^{\text{Enc}}, r_k, r_m)$ to \mathcal{P} , and $(r_k \oplus k^{\text{MAC}}, r_m \oplus m)$ to \mathcal{V} privately.

Figure 3.4: The protocol of three-party handshake.

The prover \mathcal{P} initiates the handshake by sending a regular TLS handshake request and a random nonce r_c to \mathcal{S} (in the ClientHello message). On receiving a certificate, the server nonce r_s , and a signed ephemeral DH public key $Y_S = s_S \cdot G$ from \mathcal{S} (in the ServerHello and ServerKeyExchange messages), \mathcal{P} checks the certificate and the signature and forwards them to \mathcal{V} . After performing the same check, \mathcal{V} samples a secret s_V and sends her part of the DH public key $Y_V = s_V \cdot G$ to \mathcal{P} , who then samples another secret s_P and sends the combined DH public key $Y_P = s_P \cdot G + Y_V$ to \mathcal{S} .

Since the server \mathcal{S} runs the standard TLS, \mathcal{S} will compute a DH secret as $Z = s_S \cdot Y_P$, while \mathcal{P} (and \mathcal{V}) computes its share of Z as $Z_P = s_P \cdot Y_S$ (and $Z_V = s_V \cdot Y_S$). Note that $Z = Z_P + Z_V$ where $+$ is the group operation of $EC(\mathbb{F}_p)$. Assuming the discrete logarithm problem is hard in the chosen group, Z is unknown to either party.

3.4.1.2 Step 2: key derivation

Now that \mathcal{P} and \mathcal{V} have established additive shares of Z (in the form of *EC points*), they proceed to derive session keys by evaluating the TLS-PRF [215] keyed with the x coordinate of Z .

A technical challenge here is to harmonize arithmetic operations (i.e., addition in $EC(\mathbb{F}_p)$) with bitwise operations (i.e., TLS-PRF). It is well-known that boolean circuits are not well-suited for arithmetic in large fields. As a concrete estimate, an EC Point addition resulting in just the x coordinate involves 4 subtractions, one modular inversion, and 2 modular multiplications. An estimate of the AND complexity based on the highly optimized circuits of [98] results in

over 900,000 AND gates just for the subtractions, multiplications, and modular reductions—not even including inversion, which would require running the Extended Euclidean algorithm inside a circuit.

Due to the prohibitive cost of adding EC points in a boolean circuit, \mathcal{P} and \mathcal{V} first convert the additive shares of an EC point in $EC(\mathbb{F}_p)$ to additive shares of its x -coordinate in \mathbb{F}_p , using the ECtF protocol to be presented shortly. Then the boolean circuit just involves adding two numbers in \mathbb{F}_p , using the result to key the TLS-PRF, which is much less costly than with additive shares of an EC point.

ECtF: Converting shares in $EC(\mathbb{F}_p)$ to shares in \mathbb{F}_p . The inputs to an ECtF protocol are two EC points $P_1, P_2 \in EC(\mathbb{F}_p)$. Let $P_i := (x_i, y_i)$ for $i \in \{1, 2\}$. Suppose $(x_s, y_s) = P_1 + P_2$ where $+$ is the EC group operation, the output of the protocol is $\alpha, \beta \in \mathbb{F}_p$ such that $\alpha + \beta = x_s$. Specifically, for the curve we consider, $x_s = \lambda^2 - x_1 - x_2$ where $\lambda = (y_2 - y_1)/(x_2 - x_1)$. Shares of the y -coordinate can be computed similarly but we omit that since TLS only uses the x -coordinate.

ECtF uses a Multiplicative-to-Additive (MtA) share-conversion protocol as a building block where two parties convert multiplicative shares of a secret to additive ones. We use $\alpha, \beta := \text{MtA}(a, b)$ to denote a run of MtA between Alice and Bob with inputs a and b respectively. At the end of the run, Alice and Bob receive α and β such that $a \cdot b = \alpha + \beta$. Unless otherwise specified, all arithmetic operations are in some finite field \mathbb{F}_p . The protocol can be generalized to convert multiple multiplicative shares to additive shares of their inner product without increasing the communication complexity. Namely for vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_p^n$, if $\alpha, \beta := \text{MtA}(\mathbf{a}, \mathbf{b})$, then $\langle \mathbf{a}, \mathbf{b} \rangle = \alpha + \beta$. See, e.g., [128] for a Paillier [202]-based construction.

Now we specify the protocol of ECtF. ECtF has two main ingredients. Let $[a]$ denote a 2-out-of-2 sharing of a , i.e., $[a] = (a_1, a_2)$ such that party i has a_i for $i \in \{1, 2\}$ while $a = a_1 + a_2$. The first ingredient is share inversion: given $[a]$, compute $[a^{-1}]$. As shown in [128], we can use the inversion protocol of Bar-Ilan and Beaver [44] together with MtA as follows: party i samples a random value r_i and executes MtA to compute $\delta_1, \delta_2 := \text{MtA}((a_1, r_1), (r_2, a_2))$. Note that $\delta_1 + \delta_2 = a_1 \cdot r_2 + a_2 \cdot r_1$. Party i publishes $v_i = \delta_i + a_i \cdot r_i$ and thus both parties learn $v = v_1 + v_2$. Finally, party i outputs $\beta_i = a_i \cdot v^{-1}$. The protocol computes a correct sharing of a^{-1} because $\beta_1 + \beta_2 = a^{-1}$. Moreover, the protocol doesn't leak a to any party assuming MtA is secure. In fact, party i 's view consists of $(a_1 + a_2)(r_1 + r_2)$, which is uniformly random since r_i is uniformly random.

The second ingredient is share multiplication: compute $[ab]$ given $[a], [b]$. $[ab]$ can be computed using MtA as follows: parties execute MtA to compute $\alpha_1, \alpha_2 := \text{MtA}((a_1, b_1), (b_2, a_2))$. Note that $\alpha_1 + \alpha_2 = a_1 \cdot b_2 + a_2 \cdot b_1$. Finally, party i outputs $m_i = \alpha_i + a_i \cdot y_i$. The security and correctness of the protocol can be argued similarly as above.

Combining these two ingredients, Fig. 3.5 presents the ECtF protocol, with communication complexity 8 ciphertexts.

Secure evaluation of the TLS-PRF. Having computed shares of the x -coordinate of Z , the so called premaster secret in TLS, in ECtF, \mathcal{P} and \mathcal{V} evaluate the TLS-PRF in 2PC to derive session keys. Beginning with the SHA-256 circuit of [69], we hand-optimized the TLS handshake circuit resulting in a circuit with total AND complexity of 779,213.

ECTf between \mathcal{P} and \mathcal{V}

Input: $P_1 = (x_1, y_1) \in EC(\mathbb{F}_p)$ from \mathcal{P} , $P_2 = (x_2, y_2) \in EC(\mathbb{F}_p)$ from \mathcal{V} .

Output: \mathcal{P} and \mathcal{V} output s_1 and s_2 such that $s_1 + s_2 = x$ where $(x, y) = P_1 + P_2$ in EC .

Protocol:

- \mathcal{P} (and \mathcal{V}) sample $\rho_i \leftarrow \mathbb{Z}_p$ for $i \in \{1, 2\}$ respectively. \mathcal{P} and \mathcal{V} run $\alpha_1, \alpha_2 := \text{MtA}((-x_1, \rho_1), (\rho_2, x_2))$.
- \mathcal{P} computes $\delta_1 = -x_1\rho_1 + \alpha_1$ and \mathcal{V} computes $\delta_2 = x_2\rho_2 + \alpha_2$.
- \mathcal{P} (and \mathcal{V}) reveal δ_1 (and δ_2) to each other and compute $\delta = \delta_1 + \delta_2$.
- \mathcal{P} (and \mathcal{V}) compute $\eta_i = \rho_i \cdot \delta^{-1}$ for $i \in \{1, 2\}$ respectively.
- \mathcal{P} and \mathcal{V} run $\beta_1, \beta_2 := \text{MtA}((-y_1, \eta_1), (\eta_2, y_2))$.
- \mathcal{P} computes $\lambda_1 = -y_1 \cdot \eta_1 + \beta_1$ and \mathcal{V} computes $\lambda_2 = y_2 \cdot \eta_2 + \beta_2$. They run $\gamma_1, \gamma_2 := \text{MtA}(\lambda_1, \lambda_2)$.
- \mathcal{P} (and \mathcal{V}) computes $s_i = 2\gamma_i + \lambda_i^2 - x_i$ for $i \in \{1, 2\}$ respectively.
- \mathcal{P} outputs s_1 and \mathcal{V} outputs s_2 .

Figure 3.5: The protocol for converting shares of EC points in $EC(\mathbb{F})$ to shares of coordinates in \mathbb{F} .

3.4.1.3 Adapting to support GCM and TLS 1.3

GCM. For GCM, a single key (for each direction) is used for both encryption and MAC. Adapting the above protocol to support GCM in TLS 1.2 is straightforward. The first step would remain identical, while output of the second step needs to be truncated, as GCM keys are shorter. The adaption to TLS 1.3 is discussed below.

TLS 1.3. The specification of TLS 1.3 [214] has been recently published. We refer readers to Section 3.2 for the differences from TLS 1.2. To support TLS 1.3, the 3P-HS protocol must be adapted to a new handshake flow and a different key derivation circuit.

Notably, all handshake messages after the ServerHello are now encrypted.

A naïve strategy would be to decrypt them in 2PC, which would be costly as certificates are usually large. However, thanks to the key independence property of TLS 1.3 [106], \mathcal{P} and \mathcal{V} can securely reveal the handshake encryption keys without affecting the secrecy of final session keys [106]. Handshake integrity is preserved because the Finished message authenticates the handshake using yet another independent key. (In fact [106, §3.1] argues that the signatures already authenticate the handshake.) Therefore the optimized 3P-HS work as follows. \mathcal{P} and \mathcal{V} perform ECDHE the same as before. Then they derive handshake and application keys by executing 2PC-HKDF, and reveal the handshake keys to \mathcal{P} , allowing \mathcal{P} to finish the handshake locally (i.e., without 2PC). The 2PC circuit complexity (involving roughly 30 invocations of SHA-256 from [69], totaling to an AND complexity of around 70k) should be comparable to that for TLS 1.2. Finally, since CBC-HMAC is not supported by TLS 1.3, DECO can only be used in GCM mode.

3.4.2 Query execution

After the handshake, the prover \mathcal{P} sends her query Q to the server \mathcal{S} as a standard TLS client, but with help from the verifier \mathcal{V} . We first focus on one-round sessions where \mathcal{P} sends all queries to \mathcal{S} before receiving any response. Most applications of DECO, e.g., proving provenance of content retrieved via HTTP GET, are one-round. Extending DECO to support multi-round sessions is discussed later.

Specifically, since session keys are secret-shared, the two parties need to execute a 2PC protocol to construct TLS records encrypting Q without divulging

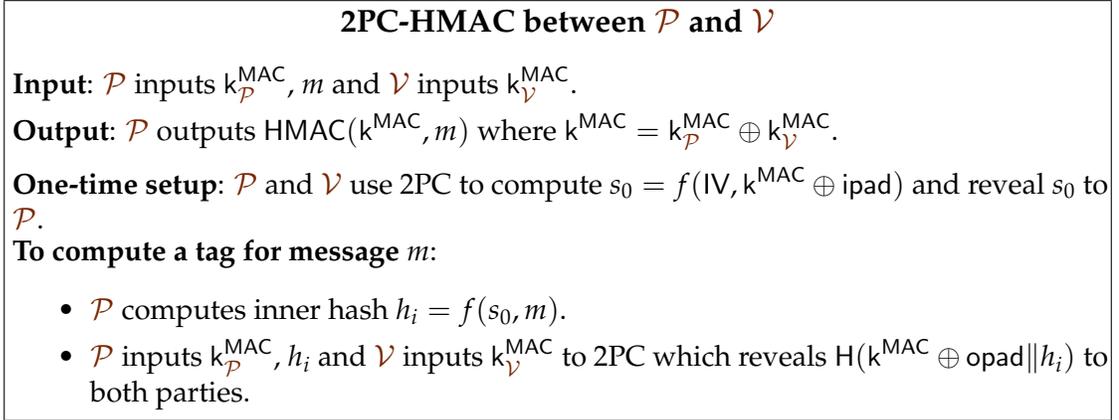


Figure 3.6: The 2PC-HMAC protocol. f denotes the compression function of the hash function H and IV denotes the initial value.

their private key shares. Although generic 2PC would in theory suffice, it would be expensive for large queries. We instead introduce custom 2PC protocols that are orders-of-magnitude more efficient.

3.4.2.1 CBC-HMAC

The protocol is specified in Fig. 3.6. Recall that \mathcal{P} and \mathcal{V} hold shares of the MAC key, while \mathcal{P} holds the encryption key. To construct TLS records encrypting Q —potentially private to \mathcal{P} , the two parties first run a 2PC protocol to compute the HMAC tag τ of Q , and then \mathcal{P} encrypts $Q || \tau$ locally and sends the ciphertext to \mathcal{S} .

Let H denote SHA-256. Recall that the HMAC of message m with key k is

$$\text{HMAC}_H(k, m) = H((k \oplus \text{opad}) || \underbrace{H((k \oplus \text{ipad}) || m)}_{\text{inner hash}}).$$

A direct 2PC implementation would be expensive for large queries, as it requires hashing the entire query in 2PC to compute the inner hash. The key idea in our optimization is to make the computation of the inner hash local to \mathcal{P} (i.e.,

without 2PC). If \mathcal{P} knew $k \oplus \text{ipad}$, she could compute the inner hash. We cannot, though, simply give $k \oplus \text{ipad}$ to \mathcal{P} , as she could then learn k and forge MACs.

Our optimization exploits the Merkle–Damgård structure in SHA-256. Suppose m_1 and m_2 are two correctly sized blocks. Then $H(m_1 \| m_2)$ is computed as $f_H(f_H(\text{IV}, m_1), m_2)$ where f_H denotes the one-way compression function of H , and IV the initial vector.

After the three-party handshake, \mathcal{P} and \mathcal{V} execute a simple 2PC protocol to compute $s_0 = f_H(\text{IV}, k^{\text{MAC}} \oplus \text{ipad})$, and reveal it to \mathcal{P} . To compute the inner hash of a message m , \mathcal{P} just uses s_0 as the IV to compute a hash of m . Revealing s_0 does not reveal k^{MAC} , as f_H is assumed to be one-way. To compute $\text{HMAC}(m)$ then involves computing the outer hash in 2PC on the inner hash, a much shorter message. Thus, we manage to reduce the amount of 2PC computation to a few blocks regardless of query length, as opposed to up to 256 blocks with generic 2PC (recall that computing the HMAC of a 16KB record involves around 256 invocations of f_H .)

3.4.2.2 AES-GCM

For GCM, \mathcal{P} and \mathcal{V} perform authenticated encryption of Q . 2PC-AES is straightforward with optimized circuits (e.g., [28]), but computing tags for large queries is expensive as it involves evaluating long polynomials in a large field *for each record*. Our optimized protocol makes polynomial evaluation local via precomputation. We refer readers to Appendix B.1.2 for details. Since 2PC-GCM involves not only tag creation but also AES encryption, it incurs higher computational cost and latency than CBC-HMAC.

3.4.3 Extensions

The above protocols suffice for most DECO applications, but some extensions can be useful.

Optionality of query construction protocols. For applications that bind responses to queries, e.g., when a stock ticker is included with the quote, 2PC query construction protocols can be avoided altogether. Since TLS uses separate keys for each direction of communication, client-to-server keys can be revealed to \mathcal{P} after the handshake so that \mathcal{P} can query the server without interacting with \mathcal{V} .

Supporting multi-round sessions. DECO can be extended to support multi-round sessions where \mathcal{P} sends further queries depending on previous responses. After each round, \mathcal{P} executes similar 2PC protocols as above to verify MAC tags of incoming responses, since MAC verification and creation is symmetric. However an additional commitment is required to prevent prevent \mathcal{P} from abusing MAC verification to forge tags.

In TLS, different MAC keys are used for server-to-client and client-to-server communication. To support multi-round sessions, \mathcal{P} and \mathcal{V} run 2PC to verify tags for former, and create tags on fresh messages for latter. We've specified the protocols to create (and verify) MAC tags. Now we discuss additional security considerations for multi-round sessions.

When checking tags for server-to-client messages, we must ensure that \mathcal{P} cannot forge tags on messages that are not originally from the server. Suppose

\mathcal{P} wishes to verify a tag T on message M . The idea is to have \mathcal{P} first commit to T , then \mathcal{P} and \mathcal{V} run a 2PC protocol to compute a tag T' on message M . \mathcal{P} is asked to open the commitment to \mathcal{V} and if $T \neq T'$, \mathcal{V} aborts the protocol. Since \mathcal{P} doesn't know the MAC key, \mathcal{P} cannot compute and commit to a tag on a message that is not from the server.

When creating tags for client-to-server messages, \mathcal{V} makes sure MAC tags are created on messages with increasing sequence numbers, as required by TLS. This also prevents a malicious \mathcal{P} from creating two messages with the same sequence number, because there is no way for \mathcal{V} to distinguish which one was sent to the server.

3.4.4 Full protocol

After querying the server and receiving a response, \mathcal{P} commits to the session by sending the ciphertexts to \mathcal{V} , and receives \mathcal{V} 's MAC key share. Then \mathcal{P} can verify the integrity of the response, and prove statements about it. Figure 3.7 specifies the full DECO protocol for CBC-HMAC (the protocol for GCM is similar and described later).

For clarity, we abstract away the details of zero-knowledge proofs in an ideal functionality \mathcal{F}_{ZK} like that in [142]. On receiving (“prove”, x, w) from \mathcal{P} , where x and w are private and public witnesses respectively, \mathcal{F}_{ZK} sends w and the relationship $\pi(x, w) \in \{0, 1\}$ (defined below) to \mathcal{V} . Specifically, for CBC-HMAC, x, w, π are defined as follows: $x = (k^{\text{Enc}}, \theta_s, Q, R)$ and $w = (\hat{Q}, \hat{R}, k^{\text{MAC}}, b)$. The relationship $\pi(x, w)$ outputs 1 if and only if (1) \hat{Q} (and \hat{R}) is the CBC-HMAC ciphertext of Q (and R) under key $k^{\text{Enc}}, k^{\text{MAC}}$; (2) $\text{Query}(\theta_s) = Q$; and (3) $\text{Stmt}(R) = b$.

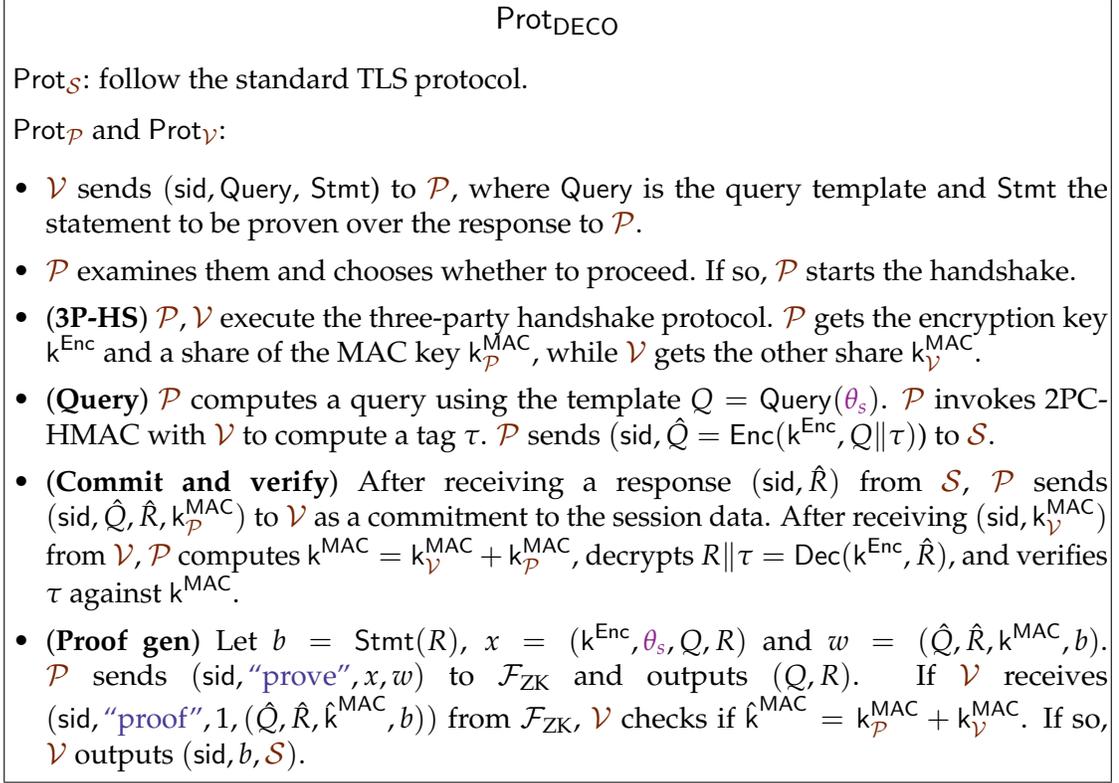


Figure 3.7: The DECO protocol. We only show the CBC-HMAC variant for clarity, while the GCM variant is described in Section 3.4.4.

Otherwise it outputs 0.

Assuming functionalities for secure 2PC and ZKPs, it can be shown that Prot_{DECO} UC-securely realizes $\mathcal{F}_{\text{Oracle}}$ for malicious adversaries, as stated in Theorem 3.1. We provide a simulation-based proof (sketch) in Appendix B.2.

Theorem 3.1 (Security of Prot_{DECO}). *Assuming the discrete log problem is hard in the group used in the three-party handshake, and that f (the compression function of SHA-256) is a random oracle, Prot_{DECO} UC-securely realizes $\mathcal{F}_{\text{Oracle}}$ in the $(\mathcal{F}_{2\text{PC}}, \mathcal{F}_{\text{ZK}})$ -hybrid world, against a static malicious adversary with abort.*

The protocol for GCM has a similar flow. We've specified the GCM variants of the three-party handshake and query construction protocols. Unlike CBC-

HMAC, GCM is not committing [139]: for a given ciphertext C encrypted with key k , one knowing k can efficiently find $k' \neq k$ that decrypts C to a different plaintext while passing the integrity check. To prevent such attacks, we require \mathcal{P} to commit to her key share $k_{\mathcal{P}}$ before learning \mathcal{V} 's key share. In the proof generation phase, in addition to proving statements about Q and R , \mathcal{P} needs to prove that the session keys used to decrypt \hat{Q} and \hat{R} are valid against the commitment to $k_{\mathcal{P}}$. Proof of the security of the GCM variant is like that for CBC-HMAC.

3.5 Proof generation

Recall that the prover \mathcal{P} commits to the ciphertext \hat{M} of a TLS session and proves to \mathcal{V} that the plaintext M satisfies certain properties. Proving only the provenance of M is easy: just reveal the encryption keys. But this sacrifices privacy. Alternatively, \mathcal{P} could prove any statement about M using general zero-knowledge techniques. But such proofs are often expensive.

In this section, we present two classes of statements optimized for what are likely to be the most popular applications: revealing only a substring of the response while proving its provenance (Section 3.5.1), or further proving that the revealed substring appears in a context expected by \mathcal{V} (Section 3.5.2).

3.5.1 Selective opening

Without loss of generality, we assume \hat{M} and M contain only one TLS record, and henceforth call them the *ciphertext record* and the *plaintext record*. Multi-record

sessions can be handled by repeating the protocol for each record.

We introduce *selective opening*, techniques that allow \mathcal{P} to efficiently *reveal* or *redact* substrings in the plaintext. Suppose the plaintext record is composed of chunks $M = (B_1, \dots, B_n)$ (details of chunking are discussed shortly). Selective opening allows \mathcal{P} to prove that the i th chunk of M is B_i , without revealing the rest of M ; we refer to this as *Reveal mode*. It can also prove that M_{-i} is the same as M but with the i th chunk removed, i.e., $M_{-i} = (B_1, \dots, B_{i-1}, B_{i+1}, \dots, B_n)$. We call this *Redact mode*. Both modes are simple, but useful for practical privacy goals. The granularity of selective opening depends on the cipher suite, which we now discuss.

3.5.1.1 CBC-HMAC

Recall that for proof generation, \mathcal{P} holds both the encryption and MAC keys k^{Enc} and k^{MAC} , while \mathcal{V} only has the MAC key k^{MAC} . Assuming SHA-256 and 128-bit AES are used, recall that a plaintext record M is first chunked into 64-byte SHA-256 blocks to compute a MAC tag then chunked into 16-byte AES blocks to compute the ciphertext \hat{M} .

Revealing a TLS record. A naïve way to prove that a ciphertext record \hat{M} encrypts a plaintext record M without revealing k^{Enc} is to prove correct encryption of each AES block in ZKP. However, this would require up to 1027 invocations of AES in ZKP (a TLS record may contain up to 16KB plaintext, i.e., 1024 AES blocks, and 3 blocks of tag), resulting in impractical performance.

Leveraging the MAC-then-encrypt structure, the same can be done using

only 3 invocations of AES in ZKP. The idea is to prove that the last few blocks of \hat{M} encrypt a tag σ and reveal the plaintext directly. Specifically, \mathcal{P} computes

$$\pi = \text{ZK-PoK}\{k^{\text{Enc}} : B_{1025} \| B_{1026} \| B_{1027} = \text{CBC}(k^{\text{Enc}}, \sigma)\}$$

and sends (M, π) to \mathcal{V} . Then \mathcal{V} verifies π and checks the MAC tag over M (note that \mathcal{V} knows the MAC key.) Its security relies on the collision-resistance of the underlying hash function in HMAC, i.e., \mathcal{P} cannot find $M' \neq M$ with the same tag σ .

Revealing a record with redacted blocks. Suppose \mathcal{P} would like to redact the i th block, i.e., to prove that $B_{i-} = (B_1, \dots, B_{i-1})$ and $B_{i+} = (B_{i+1}, \dots, B_n)$ form the prefix and suffix of the plaintext encrypted by \hat{M} . A direct proof strategy is to compute $\text{ZK-PoK}\{k^{\text{Enc}}, B_i : B_{1025} \| B_{1026} \| B_{1027} = \text{CBC}(k^{\text{Enc}}, \sigma) \wedge \sigma = \text{HMAC}(k^{\text{MAC}}, B_{i-} \| B_i \| B_{i+})\}$. In total this would cost 3 AES and 256 SHA-256 hashes in ZKP.

Leveraging the chained structure of HMAC, \mathcal{P} can redact either a prefix or a suffix from the record (depending on the position of the block), thus bringing down the cost by half in expectation. When a suffix B_{i+} is to be redacted, \mathcal{P} computes $\pi = \text{ZK-PoK}\{B_{i+}, k^{\text{Enc}} : f(s_i, B_{i+}) = ih \wedge H(k^{\text{MAC}} \oplus \text{opad} \| ih) = \sigma \wedge B_{1025} \| B_{1026} \| B_{1027} = \text{CBC}(k^{\text{Enc}}, \sigma)\}$ where $f : \{0, 1\}^{512} \rightarrow \{0, 1\}^{256}$ is the compression function of SHA-256 and s_i is the state after applying f on $B_{i-} \| B_i$. \mathcal{P} sends $(\pi, B_{i-} \| B_i)$ to \mathcal{V} . The verifier then 1) checks s_{i-1} by applying f on $B_{i-} \| B_i$, and 2) verifies π . Essentially, the security of this follows from pre-image resistance of f . Moreover, \mathcal{V} doesn't learn the redacted block since f is one-way and the output of f on it (the inner hash) is unknown. The total cost is 3 AES and $256 - i$ SHA-2 hashes in ZKP.

The protocol for redacting prefix is similar with the key difference that k^{MAC} is not revealed to \mathcal{V} . We leave the details to the full version.

Revealing a block. In CBC-HMAC, revealing a single block of a TLS record is rather expensive, because the MAC tag is computed over the entire plaintext record. Suppose \mathcal{P} would like to reveal the i th block. \mathcal{P} needs to reveal the state of applying f on B_{i-} , denoted s_{i-1} ; then prove that σ is the final state of applying f on B_{i+} with initial state s_i . \mathcal{V} verifies that $s_i = f(s_{i-1}, B_i)$. Thus, the total cost is 3 AES and $256 - i$ SHA-256 hashes in ZKP.

3.5.1.2 GCM

Unlike CBC-HMAC, revealing a block is very efficient in GCM. First, \mathcal{P} reveals $\text{AES}(k, IV)$ and $\text{AES}(k, 0)$, with proofs of correctness in ZK, to allow \mathcal{V} to verify the integrity of the ciphertext. Then, to reveal the i th block, \mathcal{P} just reveals the encryption of the i th counter $C_i = \text{AES}(k, \text{inc}^i(IV))$ with a correctness proof. \mathcal{V} can decrypt the i th block as $\hat{B}_i \oplus C_i$. IV is the public initial vector for the session, and $\text{inc}^i(IV)$ denotes incrementing IV for i times (the exact format of inc is immaterial.) To reveal a TLS record, \mathcal{P} repeat the above protocol for each block. We defer details to Appendix B.1.3.

In summary, CBC-HMAC allows efficient selective revealing at the TLS record-level and redaction at block level in DECO, while GCM allows efficient revealing at block level. Selective opening can also serve as pre-processing to reduce the input length for a subsequent zero-knowledge proof, which we will illustrate in Section 3.6 with concrete applications.

3.5.2 Context integrity by two-stage parsing

For many applications, the verifier \mathcal{V} may need to verify that the revealed substring appears in the right context. We refer to this property as *context integrity*. In this section we present techniques for \mathcal{V} to specify contexts and for \mathcal{P} to prove context integrity efficiently.

For ease of exposition, our description below focuses on the revealing mode, i.e., \mathcal{P} reveals a substring of the server’s response to \mathcal{V} . We discuss how redaction works in Section 3.5.2.3.

3.5.2.1 Specification of contexts

Our techniques for specifying contexts assume that the TLS-protected data sent to and from a given server \mathcal{S} has a well-defined context-free grammar \mathcal{G} , known to both \mathcal{P} and \mathcal{V} . In a slight abuse of notation, we let \mathcal{G} denote both a grammar and the language it specifies. Thus, $R \in \mathcal{G}$ denotes a string R in the language given by \mathcal{G} . We assume that \mathcal{G} is *unambiguous*, i.e., every $R \in \mathcal{G}$ has a unique associated parse-tree T_R . JSON and HTML are examples of two widely used languages that satisfy these requirements, and are our focus here.

When \mathcal{P} then presents a substring R_{open} of some response R from \mathcal{S} , we say that R_{open} has *context integrity* if R_{open} is produced in a certain way expected by \mathcal{V} . Specifically, \mathcal{V} specifies a set S of positions in which she might expect to see a valid substring R_{open} in R . In our definition, S is a set of paths from the root in a parse-tree defined by \mathcal{G} to internal nodes. Thus $s \in S$, which we call a *permissible path*, is a sequence of non-terminals. Let ρ_R denote the root of T_R (the parse-tree of R in \mathcal{G}). We say that a string R_{open} has context-integrity with respect to (R, S)

if T_R has a subtree whose leaves *yield* (i.e. concatenate to form) the string R_{open} , and that there is a path $s \in S$ from ρ_R to the root of the said subtree.

Formally, we define context integrity in terms of a predicate CTX_G in Definition 3.2. At a high level, our definition is reminiscent of the production-induced context in [223].

Definition 3.2. *Given a grammar \mathcal{G} on TLS responses, $R \in \mathcal{G}$, a substring R_{open} of R , a set S of permissible paths, we define a **context function** CTX_G as a boolean function such that $CTX_G : (S, R, R_{open}) \mapsto \text{true}$ iff \exists a sub-tree $T_{R_{open}}$ of T_R with a path $s \in S$ from ρ_{T_R} to $\rho_{T_{R_{open}}}$ and $T_{R_{open}}$ yields R_{open} . R_{open} is said to have **context integrity** with respect to (R, S) if $CTX_G(S, R, R_{open}) = \text{true}$.*

As an example, consider the JSON string J in Fig. 3.3. JSON contains (roughly) the following rules:

```

Start → object           object → { pairs }
pair  → ‘key’ : value    pairs → pair | pair, pairs
key   → chars           value  → chars | object

```

In that example, \mathcal{V} was interested in learning the derivation of the pair $p_{balance}$ with key “balance” in the object given by the value of the pair $p_{checking}$ with key “checking a/c”. Each of these non-terminals is the label for a node in the parse-tree T_J . The path from the root Start of T_J to $p_{checking}$ requires traversing a sequence of nodes of the form $\text{Start} \rightarrow \text{object} \rightarrow \text{pairs}^* \rightarrow p_{checking}$, where pairs^* denotes a sequence of zero or more pairs. So S is the set of such sequences and R_{open} is the string “checking a/c”: {“balance”: \$2000}.

3.5.2.2 Two-stage parsing

Generally, proving R_{open} has context integrity, i.e., $\text{CTX}_{\mathcal{G}}(S, R, R_{\text{open}}) = \text{true}$, without directly revealing R would be expensive, since computing $\text{CTX}_{\mathcal{G}}$ may require computing T_R for a potentially long string R . However, we observed that under certain assumptions that TLS-protected data generally satisfies, much of the overhead can be removed by having \mathcal{P} *pre-process* R by applying a transformation Trans agreed upon by \mathcal{P} and \mathcal{V} , and prove that R_{open} has context integrity with respect to R' (a usually much shorter string) and S' (a set of permissible paths specified by \mathcal{V} based on S and Trans).

Based on this observation, we introduce a *two-stage parsing scheme* for efficiently computing R_{open} and proving $\text{CTX}_{\mathcal{G}}(S, R, R_{\text{open}}) = \text{true}$. Suppose \mathcal{P} and \mathcal{V} agree upon \mathcal{G} , the grammar used by the web server, and a transformation Trans . Let \mathcal{G}' be the grammar of strings $\text{Trans}(R)$ for all $R \in \mathcal{G}$. Based on Trans , \mathcal{V} specifies permissible paths S' and a constraint-checking function $\text{cons}_{\mathcal{G}, \mathcal{G}'}$. In the first stage, \mathcal{P} : (1) computes a substring R_{open} of R by parsing R (such that $\text{CTX}_{\mathcal{G}}(S, R, R_{\text{open}}) = \text{true}$) (2) computes another string $R' = \text{Trans}(R)$. In the second stage, \mathcal{P} proves to \mathcal{V} in zero-knowledge that (1) $\text{cons}_{\mathcal{G}, \mathcal{G}'}(R, R') = \text{true}$ and (2) $\text{CTX}_{\mathcal{G}'}(S', R', R_{\text{open}}) = \text{true}$. Note that in addition to public parameters $\mathcal{G}, \mathcal{G}', S, S', \text{Trans}, \text{cons}_{\mathcal{G}, \mathcal{G}'}$, the verifier only sees a commitment to R , and finally, R_{open} .

This protocol makes the zero-knowledge computation significantly less expensive by deferring actual parsing to a non-verifiable computation. In other words, the computation of $\text{CTX}_{\mathcal{G}'}(S', R', R_{\text{open}})$ and $\text{cons}_{\mathcal{G}, \mathcal{G}'}(R, R')$ can be much more efficient than that of $\text{CTX}_{\mathcal{G}}(S, R, R_{\text{open}})$.

We formalize the correctness condition for the two-stage parsing in an operational semantics rule in Definition 3.3. Here, $\langle f, \sigma \rangle$ denotes applying a function f on input σ , while $\frac{P}{C}$ denotes that if the premise P is true, then the conclusion C is true.

Definition 3.3. *Given a grammar \mathcal{G} , a context function and permissible paths $\text{CTX}_{\mathcal{G}}(S, \cdot, \cdot)$, a transformation Trans , a grammar $\mathcal{G}' = \{R' : R' = \text{Trans}(R), R \in \mathcal{G}\}$ with context function and permissible paths $\text{CTX}_{\mathcal{G}'}(S', \cdot, \cdot)$ and a function $\text{cons}_{\mathcal{G}, \mathcal{G}'}$, we say $(\text{cons}_{\mathcal{G}, \mathcal{G}'}, S')$ are correct w.r.t. S , if for all (R, R', R_{open}) such that $R \in \mathcal{G}$, booleans b the following rule holds:*

$$\frac{\langle \text{cons}_{\mathcal{G}, \mathcal{G}'}, (R, R') \rangle \Rightarrow \text{true} \quad \langle \text{CTX}_{\mathcal{G}'}, (S', R', R_{\text{open}}) \rangle \Rightarrow b}{\langle \text{CTX}_{\mathcal{G}}, (S, R, R_{\text{open}}) \rangle \Rightarrow b}.$$

Below, we focus on a grammar that most DECO applications use, and present concrete constructions of two-stage parsing schemes.

3.5.2.3 DECO focus: Key-value grammars

A broad class of data formats, such as JSON, have a notion of key-value pairs. Thus, they are our focus in the current version of DECO.

A key-value grammar \mathcal{G} produces key-value pairs according to the rule, “pair \rightarrow start key middle value end”, where start, middle and end are delimiters. For such grammars, an array of optimizations can greatly reduce the complexity for proving context. We discuss a few such optimizations below, with formal specification relegated to Appendix B.3.

Revelation for a globally unique key. For a key-value grammar \mathcal{G} , set of paths S , if for an $R \in \mathcal{G}$, a substring R_{open} satisfying context-integrity requires that

R_{open} is parsed as a key-value pair with a globally unique key K (formally defined in Appendix B.3.4), R_{open} simply needs to be a substring of R and correctly be parsed as a pair. Specifically, $\text{Trans}(R)$ outputs a substring R' of R containing the desired key, i.e., a substring of the form “start K middle value end” and \mathcal{P} can output $R_{\text{open}} = R'$. \mathcal{G}' can be defined by the rule $S_{\mathcal{G}'} \rightarrow \text{pair}$ where $S_{\mathcal{G}'}$ is the start symbol in the production rules for \mathcal{G}' . Then (1) $\text{cons}_{\mathcal{G},\mathcal{G}'}(R, R')$ checks that R' is a substring of R and (2) for $S' = \{S_{\mathcal{G}'}\}$, $\text{CTX}_{\mathcal{G}'}(S', R', R_{\text{open}})$ checks that (a) $R' \in \mathcal{G}'$ and (b) $R_{\text{open}} = R'$. Globally unique keys arise in Section 3.6.2 when selectively opening the response for age.

Redaction in key-value grammars. Thus far, our description of two-stage parsing assumes the Reveal mode in which \mathcal{P} reveals a substring R_{open} of R to \mathcal{V} and proves that R_{open} has context integrity with respect to the set of permissible paths specified by \mathcal{V} . In the Redact mode, the process is similar, but instead of revealing R_{open} in the clear, \mathcal{P} generates a commitment to R_{open} using techniques from Section 3.5.1 and reveals R , with R_{open} removed, for e.g. by replacing its position with a dummy character.

3.6 Applications

DECO can be used for any oracle-based application. To showcase its versatility, we have implemented and evaluated three applications that leverage its various capabilities: 1) a confidential financial instrument realized by smart contracts; 2) converting legacy credentials to anonymous credentials; and 3) privacy-preserving price discrimination reporting. Evaluation results are pre-

sented in Section 3.7.2.

3.6.1 Confidential financial instruments

Financial derivatives are among the most commonly cited smart contract applications [76, 200], and exemplify the need for authenticated data feeds (e.g., stock prices). For example, one popular financial instrument that is easy to implement in a smart contract is a *binary option* [6]. This is a contract between two parties betting on whether, at a designated future time, e.g., the close of day D , the price P^* of some asset N will equal or exceed a predetermined target price P , i.e., $P^* \geq P$. A smart contract implementing this binary option can call an oracle \mathcal{O} to determine the outcome.

In principle, \mathcal{O} can conceal the underlying asset N and target price P for a binary option on chain. It simply accepts the option details off chain, and reports only a bit specifying the outcome $\text{Stmt} := P^* \geq? P$. This approach is introduced in [159], where it is referred to as a *Mixicle*.

A limitation of a basic Mixicle construction is that \mathcal{O} itself learns the details of the financial instrument. Prior to DECO, only oracle services that use TEE (e.g., [257]) could conceal queries from \mathcal{O} . We now show how DECO can support execution of the binary option *without \mathcal{O} learning the details of the financial instrument, i.e., N or P* ².

The idea is that the option winner plays the role of \mathcal{P} , and obtains a signed result of Stmt from \mathcal{O} , which plays the role of \mathcal{V} . We now describe the protocol

²The predicate direction $\geq?$ or $\leq?$ can be randomized. Concealing winner and loser identities and payment amounts is discussed in [159]. Additional steps can be taken to conceal other metadata, e.g., the exact settlement time.

and its implementation.

Protocol. Let $\{sk_{\mathcal{O}}, pk_{\mathcal{O}}\}$ denote the oracles' key pair. In our scheme, a binary option is specified by an asset name N , threshold price P , and settlement date D . We denote the commitment of a message M by $\mathcal{C}_M = \text{com}(M, r_M)$ with a witness r_M . Figure 3.8 shows the workflow steps in a confidential binary option:

1) *Setup*: Alice and Bob agree on the binary option $\{N, P, D\}$ and create a smart contract \mathcal{SC} with identifier $ID_{\mathcal{SC}}$. The contract contains $pk_{\mathcal{O}}$, addresses of the parties, and commitments to the option $\{\mathcal{C}_N, \mathcal{C}_P, \mathcal{C}_D\}$ with witnesses known to both parties. They also agree on public parameters θ_p (e.g., the URL to retrieve asset prices).

2) *Settlement*: Suppose Alice wins the bet. To claim the payout, she uses DECO to generate a ZK proof that the current asset price retrieved matches her position. Alice and \mathcal{O} execute the DECO protocol (with \mathcal{O} acting as the verifier) to retrieve the asset price from θ_p (the target URL). We assume the response contains (N^*, P^*, D^*) . In addition to the ZK proof in DECO to prove origin θ_p , Alice proves the following statement:

$$\text{ZK-PoK}\{P, N^*, P^*, D^*, r_N, r_P, r_D : (P \leq P^*) \wedge \mathcal{C}_N = \text{com}(N^*, r_N) \wedge \mathcal{C}_P = \text{com}(P, r_P) \wedge \mathcal{C}_D = \text{com}(D^*, r_D)\}.$$

Upon successful proof verification, the oracle returns a signed statement with the contract ID, $S = \text{Sig}(sk_{\mathcal{O}}, ID_{\mathcal{SC}})$.

3) *Payout*: Alice provides the signed statement S to the contract, which verifies the signature and pays the winning party.

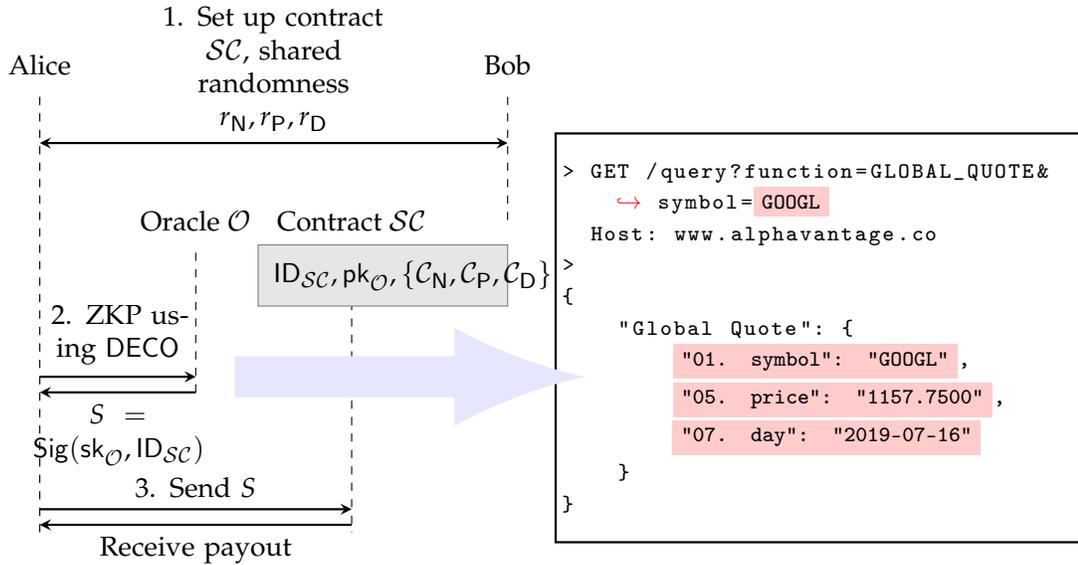


Figure 3.8: Two parties Alice and Bob execute a confidential binary option. Alice uses DECO to access a stock price API and convince \mathcal{O} she has won. Examples of request and response are shown to the right. Text in red is sensitive information to be redacted.

Alice and Bob need to trust \mathcal{O} for integrity, but not for privacy. They can further hedge against integrity failure by using multiple oracles, as explained in Section 3.3.1. Decentralizing trust over oracles is a standard and already deployed technique [113]. We emphasize that DECO ensures privacy even if all the oracles are malicious.

Implementation details. Figure 3.8 shows the request and response of a stock price API. Let \hat{R} and R denote the response ciphertext and the plaintext respectively. To settle an option, \mathcal{P} proves to \mathcal{V} that R contains evidence that he won the option, using the two-stage parsing scheme introduced in Section 3.5.2. In the first stage, \mathcal{P} parses R locally and identifies the smallest substring of R that can convince \mathcal{V} . E.g., for stock prices, $R_{\text{price}} = "05. price": "1157.7500"$ suffices. In the second stage, \mathcal{P} proves knowledge of $(R_{\text{price}}, P, r_P)$ in ZK such that

- 1) R_{price} is a substring of the decryption of \hat{R} ; 2) R_{price} starts with "05. price";
- 3) the subsequent characters form a floating point number P^* and that $P^* \geq P$;
- 4) $\text{com}(P, r_P) = C_P$.

This two-stage parsing is secure assuming the keys are unique and the key "05. price" is followed by the price, making the grammar of this response a *key-value grammar with unique keys*, as discussed in Section 3.5.2. Similarly, \mathcal{P} proves that the stock name and date contained in R match the commitments. With the CBC-HMAC ciphersuite, the zero-knowledge proof circuit involves redacting 7 SHA-2 blocks, computing commitments, and string processing.

The user (\mathcal{P}) also needs to reveal enough portion of the HTTP GET request to oracle (\mathcal{V}) in order to convince access to the correct API endpoint. The GET request contains several parameters—some to be revealed like the API endpoint, and others with sensitive details like stock name and private API key. \mathcal{P} redacts sensitive params using techniques from Section 3.5.1 and reveals the rest to \mathcal{V} . Without additional care though, a cheating \mathcal{P} can alter the semantics of the GET request and conceal the cheating by redacting extra parameters. To ensure this does not happen, \mathcal{P} needs to prove that the delimiter "&" and separator "=" do not appear in the redacted text. The security is argued below.

HTTP GET requests (and HTML) have a special restriction: the demarcation between a key and a value (i.e., middle) and the start of a key-value pair (i.e., start) are never substrings of a key or a value. This means that to redact more than a single contiguous key or value, \mathcal{P} must redact characters in $\{\text{middle}, \text{start}\}$. So we have $\text{cons}_{\mathcal{G}, \mathcal{G}'}(R, R')$ check that: (1) $|R| = |R'|$; and (2) $\forall i \in |R'|$, either $R'[i] = D \wedge R[i] \notin \{\text{middle}, \text{start}\}$ or $R[i] = R'[i]$ (D is a dummy character used to do in-place redaction). Checking $\text{CTX}_{\mathcal{G}'}$ is then unnecessary.

3.6.2 Legacy credentials to anonymous credentials: Age proof

User credentials are often inaccessible outside a service provider’s environment. Some providers offer third-party API access via OAuth tokens, but such tokens reveal user identifiers. DECO allows users holding credentials in existing systems (what we call *legacy credentials*) to prove statements about them to third parties (verifiers) *anonymously*. Thus, DECO is the first system that allows users to convert *any* web-based legacy credential into an anonymous credential without server-side support [216] or trusted hardware [257].

We showcase an example where a student proves her/his age is over 18 using credentials (demographic details) stored on a University website. A student can provide this proof of age to any third party, such as a state issuing a driver’s license or a hospital seeking consent for a medical test. Figure 3.9 shows an example university website page containing demographic information such as the name, birth date, student ID among others.

To prove age over 18, the prover parses 6-7 AES blocks that contain the birth date and proves her age is above 18 in ZK to the verifier. Like other examples, due to the unique HTML tags surrounding the birth date, this is also a key-value grammar with unique keys (see Section 3.5.2). Similar to application 1, this example requires additional string processing to parse the date and compute age.

```
<title>Demographic Data</title>
<span id='EMPLID'> 111111 </span>
<span id='NAME'> Alice </span>
<span id='BIRTHDATE'> 01/01/1990 </span> ...
```

Figure 3.9: The demographic details of a student displayed on a Univ. website. Highlighted text contains student age. Reveal mode is used together with two-stage parsing.

3.6.3 Price discrimination

Price discrimination refers to selling the same product or service at different prices to different buyers. Ubiquitous consumer tracking enables online shopping and booking websites to employ sophisticated price discrimination [243], e.g., adjusting prices based on customer zip codes [148]. Price discrimination can lead to economic efficiency [199], and is thus widely permissible under existing laws.

In the U.S., however, the FTC forbids price discrimination if it results in competitive injury [124], while new privacy-focused laws in Europe, such as the GDPR, are bringing renewed focus to the legality of the practice [262]. Consumers in any case generally dislike being subjected to price discrimination. Currently, however, there is no trustworthy way for users to report online price discrimination.

DECO allows a buyer to make a verifiable claim about perceived price discrimination by proving the advertised price of a good is higher than a threshold, while redacting sensitive information such as name and address. We implement this example using the CBC-HMAC cipher suite for the TLS session and redact 12 SHA-2 blocks containing user name and address (See Fig. 3.10).

Figure 3.10 shows parts of an order invoice page on a shopping website (Amazon) with personal details such as the name and address of the buyer.

```
<table>
<tr>Order Placed: November 23, 2018</tr>
<tr>Order Total: $34.28</tr>
<tr>Items Ordered: Food Processor</tr>
</table>
...
<b> Shipping Address: </b>
<ul class="displayAddressUL">
<li class="FullName"> Alice </li>
<li class="Address"> Wonderland </li>
<li class="City"> New York </li>
</ul>
```

Figure 3.10: The order invoice page on Amazon in HTML. Redact mode is used to redact the sensitive text in red, while the rest is revealed.

The buyer wants to convince a third-party (verifier) about the charged price of a particular product on a particular date. In this example, we demonstrate a redaction strategy where multiple personal attributes such as the name and address are redacted from the invoice. We use the CBC-HMAC for the TLS session along with the Redact mode to redact 14 blocks from the invoice. Like other examples, the HTML tags, such as “City” with the corresponding values form a key-value grammar. The start non-terminal is of the form `<li class=`, the middle non-terminal is `>` and the end non-terminal is ``. The strings “<” and “>” never occur within a key or a value, hence reducing the redaction problem to the one discussed above in Section 3.6.1.

3.7 Implementation and Evaluation

In this section, we discuss implementation details and evaluation results for DECO and our three applications.

| | | LAN | | WAN | |
|----------------|--------------|--------------|-------------|--------------|-------------|
| | | Online | Offline | Online | Offline |
| 3P-Handshake | TLS 1.2 only | 368.5 (0.6) | 1668 (4) | 2850 (20) | 10290 (10) |
| 2PC-HMAC | TLS 1.2 only | 133.8 (0.5) | 164.9 (0.4) | 2520 (20) | 3191 (8) |
| 2PC-GCM (256B) | 1.2 and 1.3 | 36.65 (0.02) | 392 (8) | 1208.5 (0.2) | 12010 (70) |
| 2PC-GCM (512B) | 1.2 and 1.3 | 53.0 (0.5) | 610 (10) | 2345 (1) | 12520 (70) |
| 2PC-GCM (1KB) | 1.2 and 1.3 | 101.9 (0.5) | 830 (20) | 4567 (4) | 14300 (200) |
| 2PC-GCM (2KB) | 1.2 and 1.3 | 204.7 (0.9) | 1480 (30) | 9093.5 (0.9) | 18500 (200) |

Table 3.1: Run time of 3P-HS and query execution protocols. All times are in milliseconds.

3.7.1 DECO protocols

We implemented the three-party handshake protocol (3P-HS) for TLS 1.2 and query execution protocols (2PC-HMAC and 2PC-GCM) in about 4700 lines of C++ code. We built a hand-optimized TLS-PRF circuit with total AND complexity of 779,213. We also used variants of the AES circuit from [28]. Our implementation uses the relic toolkit [34], the EMP toolkit [246], and the maliciously secure 2PC protocol of [247].

We integrated the three-party handshake and 2PC-HMAC protocols with mbedTLS [35], a widely used TLS implementation, to build an end-to-end system. 2PC-GCM can be integrated to TLS similarly with more engineering effort. We evaluated the performance of 2PC-GCM separately. The performance impact of integration should be negligible. We did not implement 3P-HS for TLS 1.3, but we conjecture the performance should be comparable to that for TLS 1.2, since the the circuit complexity is similar (c.f. Section 3.4.1.3).

Evaluation. We evaluated the performance of DECO protocols in both the LAN and WAN settings. Both the prover and verifier run on a c5.2xlarge AWS node with 8 vCPU cores and 16GB of RAM. We located the two nodes in the same

region (but different availability zones) for the LAN setting, but in two distinct data centers (in Ohio and Oregon) in the WAN setting. The round-trip time between two nodes in the LAN and WAN is about 1ms and 67ms, respectively.

Table 3.1 summarizes the runtime of DECO protocols during a TLS session. 50 samples were used to compute the mean and standard error of the mean (in parenthesis). The MPC protocol we used relies on offline preprocessing to improve performance. Since the offline phase is input- and target-independent, it can be done prior to the TLS session. Only the online phase is on the critical path.

As shown by the evaluation, DECO protocols are very efficient in the LAN setting. It takes 0.37 seconds to finish the three-party handshake. For query execution, 2PC-HMAC is efficient (0.13s per record) as it only involves one SHA-2 evaluation in 2PC, regardless of record size. 2PC-GCM is generally more expensive and the cost depends on the query length, as it involves 2PC-AES over the entire query. We evaluated its performance with queries ranging from 256B to 2KB, the typical sizes seen in HTTP GET requests [211]. In the LAN setting, the performance is efficient and comparable to 2PC-HMAC.

In the WAN setting, the runtime is dominated by the network latency because MPC involves many rounds of communication. Nonetheless, the performance is still acceptable, given that DECO is likely to see only periodic use for most applications we consider.

| | Binary Option | Age Proof | Price Discrimination |
|---------------|------------------|------------------|----------------------|
| prover time | $9.70 \pm 0.04s$ | $3.67 \pm 0.02s$ | $7.93 \pm 0.02s$ |
| verifier time | 0.01s | 0.01s | 0.01s |
| proof size | 861B | 574B | 861B |
| # constraints | 511k | 164k | 405k |
| memory | 1.40GB | 0.69GB | 0.92GB |

Table 3.2: Costs of generating and verifying ZKPs in proof-generation phase of DECO for applications in Section 3.6.

3.7.2 Proof generation

We instantiated zero-knowledge proofs with a standard proof system [49] in `libsnark` [14]. We have devised efficiently provable statement templates, but users of DECO need to adapt them to their specific applications. SNARK compilers enable such adaptation in a high-level language, concealing low-level details from developers. We used `xjsnark` [172] and its Java-like high-level language to build statement templates and `libsnark` compatible circuits.

Our rationale in choosing `libsnark` (as opposed to MPC-in-head [156] or other protocols) is its relatively mature tooling support. The proofs generated by `libsnark` are constant-size and very efficient to verify, the downside being the per-circuit trusted setup. With more effort, DECO can be adapted to use, e.g., Bulletproofs [62], which requires no trusted setup but has large proofs and verification time.

Evaluation. We measure five performance metrics for each example—prover time (the time to generate the proofs), verifier time (the time to verify proofs), proof size, number of arithmetic constraints in the circuit, and the peak memory usage during proof generation.

Table 3.2 summarizes the results. 50 samples were used to compute the mean and its standard error. Through the use of efficient statement templates and two-stage parsing, DECO achieves very practical performance. The prover time is consistently below 10s for all three applications. Since libsnark optimizes for low verification overhead, the verifier time is negligible. The number of constraints (and prover time) is highest for the binary option application due to the extra string parsing routines. We use multiple proofs in each application to reduce peak memory usage. For the most complex application, the memory usage is 1.4GB. As libsnark proofs are of a constant size 287B, the proof sizes shown are multiples of that.

3.7.3 End-to-end performance

DECO end-to-end performance depends on the available TLS ciphersuites, the size of private data, and the complexity of application-specific proofs. Here we present the end-to-end performance of the most complex application of the three we implemented—the binary option. It takes about 10.50s to finish the protocol, which includes the time taken to generate unforgeable commitments (0.50s), to run the first stage of two-stage parsing (0.30s), and to generate zero-knowledge proofs (9.70s). These numbers are computed in the LAN setting; in the WAN setting, MPC protocols are more time-consuming (5.37s), pushing the end-to-end time up to 15.37s.

In comparison, Town Crier uses TEEs to execute a similar application in about 0.6s Table 2.1, i.e., around 20x faster than DECO, *but with added trust assumptions*. Since DECO is likely to be used only periodically for most applications, its over-

head in achieving cryptographic-strength security assurances seems reasonable.

3.8 Legal and Compliance Issues

Although users can already retrieve their data from websites, DECO allows users to export the data *with integrity proofs* without their explicit approval or even awareness. We now briefly discuss the resulting legal and compliance considerations.

Critically, however, DECO *users cannot unilaterally export data* to a third party with integrity assurance, but rely on oracles as verifiers for this purpose. While DECO keeps user data private, oracles learn what websites and types of data a user accesses. Thus oracles can enforce appropriate data use, e.g., denying transactions that may result in copyright infringement.

Both users and oracles bear legal responsibility for the data they access. Recent case law on the Computer Fraud and Abuse Act (CFAA), however, shows a shift away from criminalization of web scraping [228], and federal courts have ruled that violating websites' terms of service is not a criminal act *per se* [147, 165]. Users and oracles that violate website terms of service, e.g., "click wrap" terms, instead risk *civil* penalties [39]. DECO compliance with a given site's terms of service is a site- and application-specific question.

Oracles have an incentive to establish themselves as trustworthy within smart-contract and other ecosystems. We expect that reputable oracles will provide users with menus of the particular attestations they issue and the target websites they permit, vetting these options to maximize security and minimize liability

and perhaps informing or cooperating with target servers.

The legal, performance, and compliance implications of incorrect attestations based on incorrect (and potentially subverted) data are also important. Internet services today have complex, multi-site data dependencies, though, so these issues aren't specific to DECO. Oracle services already rely on multiple data sources to help ensure correctness [113]. Oracle services in general could ultimately spawn infrastructure like that for certificates, including online checking and revocation capabilities [126] and different tiers of security [53].

3.9 Related Work

Application-layer data-provenance. Signing content at the application layer is a way to prove data provenance. However, application-layer solutions suffer from poor modularity and reusability, as they are application-specific. They also require application-layer key management, violating the principle of layer separation in that cryptographic keys are no longer confined to the TLS layer.

Cinderella [97] uses verifiable computation to convert X.509 certificates into other credential types. Its main drawback is that few users possess X.509 certificates. Open ID Connect [15] servers can issue signed claims about users and is especially promising for identity-related applications, such as [181]. However, adoption is still sparse and even among those that do support it, claims are often limited to basic information such as name and email.

Server-facilitated TLS-layer solutions. Several proposed TLS-layer data-provenance proofs [141, 59, 216] require server-side modifications. TLS-N [216] is a TLS 1.3 extension that enables a server to sign the session using the existing PKI, and also supports chunk-level redaction for privacy. We refer readers to [216] and references therein for a survey of TLS-layer solutions. Server-facilitated solutions suffer from high adoption cost, as they involve modification to security-critical server code. Moreover, they only benefit users when server administrators are able to and choose to cooperate.

Smart contract oracles. Oracles [64, 113, 257] relay authenticated data from, e.g., websites, to smart contracts. TLSNotary [23], used by Provable [27], allows a third party auditor to attest to a TLS connection between a server and a client, but relies on deprecated TLS versions (1.1 or lower). Town Crier [257] is an oracle service that uses TEEs (e.g., Intel SGX) for publicly verifiable evidence of TLS sessions and privacy-preserving computation on session data. While flexible and efficient, it relies on TEEs, which some users may reject given recently reported vulnerabilities, e.g., [61].

Selective opening with context integrity. Selective opening, i.e., decrypting part of a ciphertext to a third party while proving its integrity, has been studied previously. Sanitizable signatures [40, 60, 234, 188] allow a signed document to be selectively revealed. TLS-N [216] allows “chunk-level” redacting of TLS records. These works, however, consider a weaker adversarial model than DECO. They fail to address the critical property of context integrity. DECO enforces proofs of context integrity in the rigorous sense of Section 3.5.2, using a novel two-stage parsing scheme that achieves efficiency by greatly reducing the length

of the input to the zero-knowledge proof.

3.10 Conclusion

In this chapter, we have introduced DECO, a privacy-preserving, decentralized oracle scheme for modern TLS versions that requires no trusted hardware or server-side modifications. DECO allows users to efficiently prove provenance and fine-grained statements about session content. We also identified context-integrity attacks that are universal to privacy-preserving oracles and provided efficient mitigation in a novel two-stage parsing scheme. We formalized decentralized oracles in an ideal functionality, providing the first such rigorous security definition. DECO can liberate private data from centralized web-service silos, making it accessible to a rich spectrum of applications. We demonstrated DECO’s practicality through a fully functional implementation along with three example applications.

4.1 Introduction

Secure storage of private keys is a pervasive challenge in cryptographic systems. It is especially acute for blockchains and other decentralized systems. In these systems, private keys control the most important resources—money, identities [95], etc. Their loss has serious and often irreversible consequences.

An estimated four million Bitcoin (today worth \$14+ Billion) have vanished forever due to lost keys [217]. Many users thus store their cryptocurrency with exchanges such as Coinbase, which holds at least 10% of all circulating Bitcoin [36]. Such centralized key storage is also undesirable: It erodes the very decentralization that defines blockchain systems.

An attractive alternative is *secret sharing*. In (t, n) -secret sharing, a *committee* of n nodes holds *shares* of a secret s —usually encoded as $P(0)$ of a polynomial $P(x)$ [229]. An adversary must compromise at least $t + 1$ players to steal s , and at least $n - t$ shares must be lost to render s unrecoverable.

Proactive secret sharing (PSS), introduced in the seminal work of Herzberg et al. [145], provides even stronger security. PSS periodically *proactivizes* the shares held by players, while keeping s constant. Players obtain new shares of the secret s that are independent of their old shares, which are then discarded. Provided an adversary never obtains more than t shares between proactivizations, PSS protects the secret s against ongoing compromise of players.

Secret sharing—particularly PSS—would seem to enable users to delegate private keys safely to committees and avoid reliance on a single entity or centralized system. Indeed, a number of commercial and research blockchain systems, e.g., [37, 79, 112, 169, 263], rely on secret sharing to protect users’ keys and other sensitive data.

These systems, though, largely overlook a secret-sharing problem that is critical in blockchain systems: *node churn*.

In *permissionless* (open) blockchains, such as Bitcoin or Ethereum, nodes may freely join and leave the system at any time. In *permissioned* (closed) blockchains, only authorized nodes can join, but nodes can fail and membership change. Thus blockchain protocols for secret sharing must support committee membership changes, i.e., *dynamic* committees.

Today there are no adequate PSS schemes for dynamic committees. Existing protocols support static, but not dynamic committees [145, 67], assume weak, passive adversaries [222, 104], or incur prohibitive communication costs [249, 261, 226, 196, 45].

In this paper, we address this critical gap by introducing a new dynamic-committee proactive secret-sharing protocol called CHURP (*CHUrn-Robust Proactivation*).

4.1.1 CHURP functionality

CHURP allows a dynamic committee, i.e., one undergoing churn, to maintain a shared secret s securely.

Like a standard PSS scheme, CHURP proactivizes shares in every fixed interval of time known as an *epoch*. It supports dynamic committees as follows. An old committee of size n with a (t, n) -sharing of a secret s can transition during a *handoff* to a possibly disjoint new committee of size n with a new (t, n) -sharing of s . CHURP achieves security against an *active* adversary that compromises $t < n/2$ nodes in *each* of the old and new committees. CHURP also allows changes to t and n between epochs. (Periodic changes to s are specifically not a goal of PSS schemes, but are easy to add.)

Our main achievement in CHURP is its *very low communication complexity*: optimistic per-epoch communication complexity in a blockchain setting of $O(n)$ on-chain—which is optimal—and $O(n^2)$ off-chain, i.e., over point-to-point channels. While the on-chain complexity is lower than off-chain, it comes with the additional cost of placing transactions on the blockchain. Cheating nodes cause pessimistic $O(n^2)$ on-chain communication complexity (no off-chain cost). Both communication costs are substantially lower than in other schemes.

Despite somewhat complicated mechanics, CHURP realizes a *very simple abstraction*: It simulates a trusted third party that stores s for secure use in a wide range of applications—threshold cryptography, secure multi-party computation, etc.

4.1.2 Technical challenges and solutions

CHURP is the *first* dynamic committee PSS scheme with an end-to-end implementation that is practical even for large committees. To achieve its low communication complexity, CHURP overcomes several technical challenges in a

different manner than the prior work aimed at dynamic committees, as explained below.

The first challenge is that previous PSS schemes, relying on techniques from Herzberg et al. [145], incur high communication complexity for proactivization ($O(n^3)$ off-chain per epoch). CHURP uses a *bivariate* polynomial $B(x, y)$ to share secret s , and introduces a new proactivization protocol with cost $O(n^2)$. This protocol is based on efficient *bivariate 0-sharing*, i.e., generation of a randomized, shared polynomial $B(x, y)$ with $B(0, 0) = 0$ to refresh shares. Alternative approaches to PSS that do not explicitly generate a shared polynomial exist [132, 121], but CHURP’s 0-sharing technique is of independent interest: It can also lower the communication complexity of Herzberg et al. [145] and related schemes.

The second challenge is that during a handoff, an adversary may control t nodes in each of the old and new committees, and thus $2t$ nodes in total. Compromise of $2t$ shares in a (t, n) -sharing would leak the secret s . Previous schemes, e.g., [226], address this problem using “blinding” approaches with costly communication, while [45], address it via impractical virtualization techniques. Instead, CHURP uses a low communication-complexity technique called *dimension-switching*, that is based on known share resharing techniques. It uses an *asymmetric* bivariate polynomial $B(x, y)$, with degree t in one dimension and degree $2t$ in the other. During a handoff, it switches temporarily to a $(2t, n)$ -sharing of s to tolerate up to $2t$ compromised shares; afterward, it switches back to a (t, n) -sharing. Each switching involves a round of share resharing. Although dimension-switching is based on known techniques, CHURP’s novelty lies in applying them to the dynamic committee setting to tolerate $2t$ compromises.

Finally, most PSS schemes commit to secret degree- t polynomials using classical schemes (e.g., [118, 206]) with per-commitment size $O(t)$. CHURP uses an alternative due to Kate, Zaverucha, and Goldberg (KZG) [162] with size $O(1)$. Use of KZG for secret sharing isn't new [43], but CHURP introduces a novel KZG *hedge*. KZG assumes trusted setup and a non-standard hardness assumption. If these fail, CHURP still remains secure—but degrades to slightly weaker adversarial threshold $t < n/3$. The detection mechanisms used to hedge are efficient— $O(n)$ on-chain—and are KZG-free—so, our techniques can easily be adapted to future secret-sharing schemes that rely similarly on KZG or related non-standard assumptions.

We compose these techniques to realize CHURP with *provable security* and give a rigorous security proof.

4.1.3 Implementation and Experiments

We present an implementation of CHURP. Our experiments show very practical communication and computation costs—at least 1000x improvement over the existing state-of-the-art dynamic-committee PSS scheme [226] in the off-chain communication complexity for large committees (See Section 4.6).

Additionally, to achieve inexpensive off-chain communication among nodes in CHURP, we introduce a new technique for permissionless blockchains that is of independent interest. It leverages the peer-to-peer gossip network as a low-cost anonymous point-to-point channel. We experimentally demonstrate off-chain communication in Ethereum with monetary cost orders of magnitude less than on-chain communication.

4.1.4 Outline and Contributions

After introducing the functional, adversarial, and communication models in Section 4.2, we present our main contributions:

- *CHURN-Robust Proactive secret sharing (CHURP)*: In Section 4.3, we introduce CHURP, a dynamic-committee PSS scheme with lower communication complexity than previous schemes.
- *Novel secret-sharing techniques*: We introduce a *new 0-sharing* protocol for efficient proactivization in Section 4.4, *dimension-switching* technique to safeguard the secret in committee handoffs in Section 4.5.3, and hedging techniques for failures in the KZG commitment scheme in Section 4.5.5.
- *New point-to-point blockchain communication technique*: We introduce a novel point-to-point communication technique for permissionless blockchains in Section 4.7—usable in CHURP and elsewhere—with orders of magnitude less cost than on-chain communication.
- *Implementation and experiments*: We report on an implementation of CHURP in Section 4.6 and present performance measurements demonstrating its practicality.

We give a security proof for CHURP in Appendix C.1. We discuss related work in Section 4.9 and CHURP’s many potential applications—threshold cryptography, smart contracts with private keys, consensus simplification for light clients, etc.—in Section 4.8. We have released the CHURP system as an open-source tool at <https://www.churp.io>.

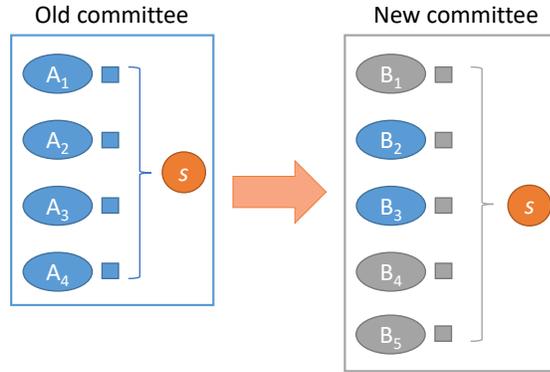


Figure 4.1: Handoff between two committees at the end of a dynamic proactive secret-sharing epoch. The secret s remains fixed. Committees may intersect, e.g., $B_2 = A_2$ and $B_3 = A_3$.

4.2 Model and Assumptions

We now describe the functional, adversarial, and communication models used for CHURP.

In a secret-sharing scheme, a *committee* of nodes shares a fixed secret s . Let \mathcal{C} denote a committee and $\{\mathcal{C}_i\}_{i=1}^n$ denote the n nodes in the committee. Each node \mathcal{C}_i holds a distinct share s_i . CHURP *proactivizes* shares, i.e., changes them periodically to prevent leakage of s to an adversary that gradually compromises nodes. Again, we emphasize that CHURP does so for a *dynamic* committee [45, 226], i.e., nodes may periodically leave / join the committee.

Shares change in a proactive secret-sharing protocol such as CHURP during what is called a *handoff* protocol. Handoff proactivizes s , i.e., changes its associated shares, while transferring s from an old committee to a new, possibly intersecting one. Fig. 4.1 depicts the handoff process. The adversarial model for proactive secret sharing in general limits adversarial control to a *threshold* t of nodes per committee. During a handoff, CHURP allows nodes to agree out of band on a change to t , as explained below.

4.2.1 Functional model

Epoch: Time in CHURP, as in any proactive secret-sharing scheme [145], is divided into fixed intervals of predetermined length called *epochs*. In each epoch, a specific committee of nodes assumes control of and then holds s . Concretely, in an epoch e , a committee $\mathcal{C}^{(e)}$ of size $N^{(e)}$ shares s using a $(t, N^{(e)})$ -threshold scheme.

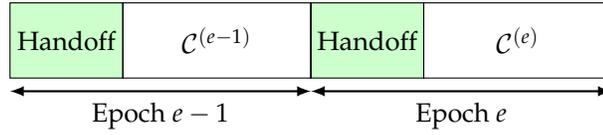


Figure 4.2: Each epoch begins with a handoff phase where the old committee hands off the secret s to the new committee. It is followed by a period of committee operation.

Handoff: Fig. 4.2 depicts the handoff at the beginning of an epoch. It involves a transfer of s from an old committee, which we denote $\mathcal{C}^{(e-1)}$, to a new committee, denoted $\mathcal{C}^{(e)}$. Prior to completion of the handoff, $\mathcal{C}^{(e-1)}$ is able to perform operations using s .

Churn: In the dynamic-committee setting of CHURP, nodes can leave a committee at any time, but can only be added during a handoff. Let $\mathcal{C}_{left}^{(e-1)}$ denote the set of nodes that have left the committee before the handoff in epoch e . Let $\mathcal{C}_{alive}^{(e-1)} = \mathcal{C}^{(e-1)} \setminus \mathcal{C}_{left}^{(e-1)}$ denote the set of nodes that participate in the handoff. We let *churn rate* α denote a bound such that $|\mathcal{C}_{alive}^{(e-1)}| \geq |\mathcal{C}^{(e-1)}|(1 - \alpha)$. Later, we provide a lower bound on the committee size using the rate α .

Keys: We assume that every node in CHURP has private / public key pair and that public keys are known to all nodes in the system. Such a setup is common in secret-sharing systems [145, 226].

4.2.2 Adversarial model

We consider a powerful active adversary \mathcal{A} . It may decide to corrupt nodes at any time. Once a node is corrupted by the adversary, it is assumed to be corrupted until the end of the current epoch. (A node may thus be “released” by an adversary in a new epoch so that it is no longer corrupted.) Corrupted nodes are allowed to deviate from the protocol arbitrarily. The proofs of correctness used by nodes in *CHURP* requires that we assume a *computationally bounded* (polynomial-time) adversary.

As noted above, we limit the adversary \mathcal{A} to corruption of no more than a threshold of nodes in a given committee. This threshold, as noted above, may change in *CHURP* through out-of-band agreement by committees. In this case, letting t and t' denote corruption thresholds for old and new committees respectively, \mathcal{A} may control at most t nodes in $\mathcal{C}^{(e-1)}$ and t' nodes in $\mathcal{C}^{(e)}$. We present the protocol in *CHURP* for threshold changes in Section 4.5.4. For simplicity of exposition, however, we assume in what follows that $t = t'$, i.e., the corruption threshold t remains fixed.

Observe that during the handoff between epochs $e - 1$ and e , members of both committees, $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$, are active. Thus \mathcal{A} may control up to $2t$ nodes at this time. As committees may intersect, i.e., an adversary may control a given node i in both the old and new committees. Alternatively, \mathcal{A} may control node i in one committee, but not the other, reflecting either a fresh corruption or node recovery.

Definition 4.1. *A protocol for dynamic-committee proactive secret sharing satisfies the following properties in the functional model above for any probabilistic polynomial*

time adversary \mathcal{A} with threshold t :

Secrecy: If \mathcal{A} corrupts no more than t nodes in a committee of any epoch, \mathcal{A} learns no information about the secret s .

Integrity: If \mathcal{A} corrupts no more than t nodes in each of the committees $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$, after the handoff, the shares for honest nodes can be correctly computed and the secret s remains intact.

4.2.3 Communication model

We aim to minimize communication complexity in CHURP. Specifically, we optimize for on-chain complexity and off-chain complexity in that order. We also consider the round complexity of our protocol designs, but prioritize communication complexity because blockchains—particularly permissionless ones—incur high costs for on-chain operations. We measure the communication complexity of our protocol (and related ones) in terms of *on-chain* and *off-chain* communication cost, as follows:

On-chain: Existing approaches such as MPSS [226] use PBFT [74] for consensus. Instead, we assume the availability of a blockchain (or other bulletin-board abstraction) to all nodes in the committee. We do this for two reasons. First, abstracting away the consensus layer results in simpler, and more modular secret-sharing protocols. Second, it makes sense to capitalize on the availability of blockchains today, rather than re-engineer their functionality.

In our model, nodes can either post a message (or) retrieve any number of messages from the blockchain. After a node posts a message to the blockchain,

within a finite time period T , it gets *published*, i.e., blockchain access is *synchronous* and the message is now retrievable by any node. This channel is assumed to be *reliable*: messages posted are not lost. This model is widely adopted in the literature (e.g., See [258, 171, 205]).

Permissionless blockchains While our techniques apply also to permissioned blockchains, we focus on permissionless blockchains—e.g., Ethereum. On such chains, users pay (heavily) for writes, but reads are free. Thus we measure on-chain communication complexity only in terms of writes, e.g., $O(n)$ on-chain cost means $O(n)$ bits written to the blockchain.

Off-chain: Nodes may alternatively communicate point-to-point (P2P) without direct use of the blockchain. We assume that every node has such a channel with every other node. P2P channels are also assumed to be *reliable*: all messages arrive without getting lost. We work in a *synchronous model*, i.e., any message sent via this channel will be received within a known bounded time period, T' .

We emphasize that synchronicity of the P2P network is required only for performance, *not* for liveness, secrecy or integrity. Looking ahead, without enough synchronicity, the off-chain protocol halts and the execution switches to the on-chain channel. In other words, an adversary may slow down the protocol execution temporarily by delaying messages, but she cannot learn or corrupt the secret. Moreover, CHURP only requires a short period of synchronicity (e.g., a few minutes) at the end of every epoch (a relatively long epoch, e.g., a day, would be the norm for CHURP). We discuss synchronicity assumptions in Section 4.5.3.3.

Off-chain P2P channels can be implemented in different ways depending

on the deployment environment. In a decentralized setting, though, nodes are often assumed not to have P2P communication, to protect them from targeted attacks and anonymity compromise. In such cases, one can use anonymous channels, such as Tor [237], to preserve anonymity with additional setup cost and engineering complexity. Alternatively, off-chain channels can be implemented by an overlay on top of the existing blockchain infrastructure. We show how to leverage the gossip network of a blockchain system [81] for inexpensive off-chain communication in Section 4.7.

We measure off-chain communication complexity as the total number of bits transmitted in P2P channels. In general, where we refer informally to proactivation protocols' *cost* in this work, we mean their communication complexity, on-chain or off-chain, as the case may be.

4.3 Overview of CHURP

Now we provide an overview of CHURP, with intuition behind our core techniques. First, we briefly review two key new techniques used in CHURP: *bivariate 0-sharing* and *dimension-switching*. (We defer details until later in the paper.) Then we give an overview and example of optimistic execution of CHURP. Finally, we briefly discuss pessimistic execution paths in CHURP, i.e., what happens when nodes are faulty, and our third key technique of hedging against failures in KZG.

4.3.1 Key secret-sharing techniques

Recall that in an ordinary (t, n) -threshold Shamir secret sharing (see [229]), shares of secret s are points on a univariate polynomial $P(x)$ such that $P(0) = s$. Instead, to enable its two key techniques, CHURP employs a *bivariate* polynomial $B(x, y)$ such that $B(0, 0) = s$. A share of $B(x, y)$ is itself a univariate polynomial: Either $B(x, i)$ or $B(i, y)$ where i is the node index.

Bivariate 0-sharing: Proactivization in nearly all secret-sharing schemes involves generating a fresh, random polynomial that shares a 0-valued secret, e.g., $Q(x, y)$ such that $Q(0, 0) = 0$. This is added to the current polynomial that encodes the secret s . We call such a polynomial $Q(x, y)$ a *0-hole* polynomial and generation of this polynomial *0-sharing*. Previous approaches' main communication bottleneck is naïve 0-sharing that incurs high ($O(n^3)$ off-chain) communication complexity. Our 0-sharing protocol achieves lower ($O(n^2)$ off-chain) complexity. (Details in Section 4.4).

Dimension-switching: CHURP uses a bivariate polynomial $B(x, y)$ asymmetric and of *non-uniform degree*. Specifically, it uses a polynomial $B(x, y)$ of degree $\langle t, 2t \rangle$. By this, we mean that it is degree- t in x (highest term x^t) and degree- $2t$ in y (highest term y^{2t}).

This structure enables our novel *dimension-switching* technique in CHURP. Nodes can switch between a sharing in the degree- t dimension of $B(x, y)$ and the degree- $2t$ dimension. The result is a change from a (t, n) -sharing of s to a $(2t, n)$ -sharing—and vice versa. We apply known resharing techniques [100, 120] via bivariate polynomials to switch between different sharings. As we show, dimension switching provides an efficient way to address a key challenge

mentioned above. During a handover, the adversary can control up to $2t$ nodes, but between handovers, we instead want a (t, n) -threshold sharing of s . (Details in Section 4.5.3.)

4.3.2 CHURP: Overview

We now give an overview of CHURP execution. We first consider the optimistic case, and discuss pessimistic cases below in Section 4.3.5.

At the end of a given epoch $e - 1$, before a handoff occurs, the current committee $\mathcal{C}^{(e-1)}$ is in what we call a *steady state*.

The committee $\mathcal{C}^{(e-1)}$ holds a (t, n) -sharing of $s = B(0, 0)$. This sharing uses the degree- t dimension of $B(x, y)$, as noted above. Node $\mathcal{C}_i^{(e-1)}$ holds share $s_i = B(i, y)$, and can compute $B(x, 0)$ for $x = i$. So it is easy to see that s_i is actually a share in a (t, n) -sharing of $B(0, 0)$. We refer to the shares in steady state as *full shares*.

During the handoff in epoch e , nodes in the old and new committees $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$ switch their sharing of s to the degree- $2t$ dimension of $B(x, y)$, resulting in what we call *reduced shares*.

Specifically, node $\mathcal{C}_j^{(e)}$ holds share $s_j = B(x, j)$. Node $\mathcal{C}_j^{(e)}$ can compute $B(0, y)$ for $y = j$, and consequently s_j is a share in a $(2t, n)$ -sharing of $B(0, 0)$. The share s_j here has “reduced” power in the sense that $2t + 1$ of these shares (as opposed to $t + 1$ full shares in steady state) are needed to reconstruct s . Thus the adversary cannot recover s despite potentially compromising $2t$ nodes across the old and new committees $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$.

After share reduction, the polynomial $B(x, y)$ is *proactivized*. A 0-hole bivariate polynomial $Q(x, y)$, i.e., such that $Q(0, 0) = 0$, is generated (using the new protocol given in Section 4.4). $Q(x, y)$ is then added to $B(x, y)$, yielding a fresh polynomial $B'(x, y) = B(x, y) + Q(x, y)$. Nodes update their reduced shares accordingly. Because $Q(x, y)$ is 0-hole, the secret s remains unchanged, i.e., $s = B'(0, 0)$.

Shares in $B'(x, y)$, i.e., for the new committee, are now *independent of those for $B(x, y)$* , i.e., for the old committee. So it is now safe to perform *full-share distribution*, i.e., to switch to the degree- t dimension of $B'(x, y)$. This involves distributing full shares to the new committee $\mathcal{C}^{(e)}$. At this point, the steady state is achieved for epoch e . Committee $\mathcal{C}^{(e)}$ holds a (t, n) -sharing of s using $B'(x, y)$.

To summarize, the three phases in the CHURP handoff are:

- *Share reduction*: Nodes switch from the degree- t dimension of $B(x, y)$ to the degree- $2t$ dimension. As a result, each node $\mathcal{C}_j^{(e)}$ in the new committee obtains a reduced share $B(x, j)$.
- *Proactivization*: The new committee generates $Q(x, y)$ such that $Q(0, 0) = 0$, and each node $\mathcal{C}_j^{(e)}$ obtains a reduced share: $B'(x, j) = B(x, j) + Q(x, j)$. Proactivization ensures that shares in the new committee are independent of those in the old.
- *Full-share distribution*: New shares $B'(i, y)$ are generated from reduced shares $\{B'(x, j)\}_j$, by switching back to the degree- t dimension of $B'(x, y)$.

The protocol thus returns to its steady state. Note that during the handoff, remaining nodes in old committee can still perform operations using s . So there is no operational discontinuity in CHURP.

4.3.3 An example

In Fig. 4.3, we show a simple example of the handoff protocol in CHURP assuming all nodes are honest. The old committee consists of three nodes $\mathcal{C}^{(e-1)} = \{A_1, A_2, A_3\}$. A_3 leaves at the end of the epoch, and a new node A'_3 joins. The new committee is thus $\mathcal{C}^{(e)} = \{A_1, A_2, A'_3\}$. The underlying polynomial $B(x, y)$ is thus of degree $\langle 1, 2 \rangle$. Node A_i 's share is $B(i, y)$ or 3 points: $B(i, 1), B(i, 2)$ and $B(i, 3)$. The figure depicts the three phases of the handoff, as follows.

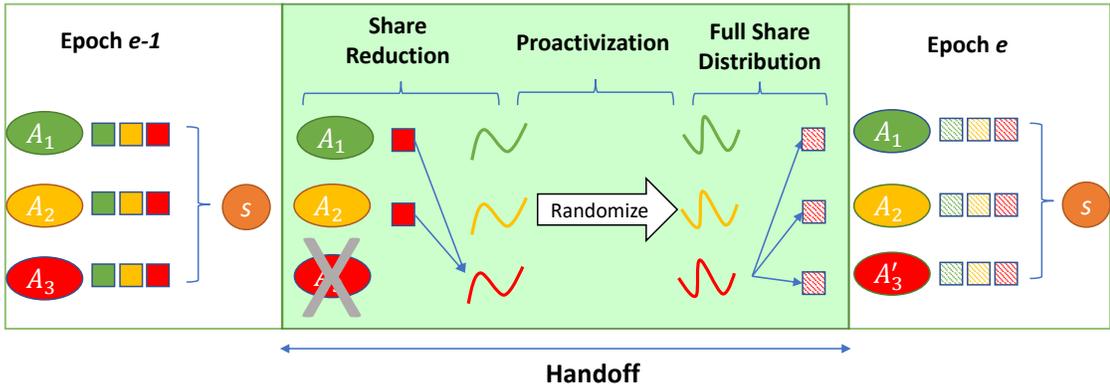


Figure 4.3: An example of the handoff protocol: Curves denote univariate polynomials (reduced shares) while squares denote points on these polynomials. See Section 4.3.3 for a description.

Share reduction: To start the handoff, each node j in the new committee constructs its reduced share $B(x, j)$ from points received from $\mathcal{C}^{(e-1)}$. As shown in the figure, node A'_3 receives points $B(1, 3)$ and $B(2, 3)$ from A_1 and A_2 respectively, from which $B(x, 3)$ can be constructed. Similarly, A_1 and A_2 construct $B(x, 1)$ and $B(x, 2)$.

Proactivization: Having reconstructed reduced shares $\{B(x, j)\}_j$, nodes in the new committee collectively generate a 0-hole bivariate polynomial $Q(x, y)$ of degree $\langle t, 2t \rangle$, with the constraint that each j only learns $Q(x, j)$. Reduced shares

are updated as $B'(x, j) = B(x, j) + Q(x, j)$. In the example above, node j ends up with $Q(x, j)$ of a random 0-hole polynomial $Q(x, y)$.

Full-share distribution: Nodes in the new committee get their full shares from the updated reduced shares. Take A_1 as an example. By this point, A_1 has $B'(x, 1)$ and sends $B'(i, 1)$ to A_i for $i \in \{2, 3\}$. Other nodes do the same. Hence, A_1 receives $B'(1, 2)$ and $B'(1, 3)$ from A_2 and A_3 respectively. It now has the necessary three points $\{B'(1, j)\}_{j \in [3]}$ in order to interpolate its full share $B'(1, y)$.

4.3.4 Active security

As noted before, the above example assumes an honest-but-curious adversary. Additional machinery in the form of cryptographic proofs of correctness for node communications—detailed in Section 4.5.3—are required against an active adversary. These proofs do not alter the overall structure of the protocol.

4.3.5 Pessimistic CHURP execution paths

What we have described thus far is an optimistic execution of CHURP. This corresponds to a subprotocol Opt-CHURP that is highly efficient and optimistic: it only completes when all nodes are honest and the assumptions of the KZG scheme hold.

When things go wrong, CHURP can detect the violation and resort to pessimistic paths. Specifically, Exp-CHURP-A can hold malicious nodes accountable. Moreover, CHURP introduces a *novel* hedge against any soundness failure of

the KZG scheme, due to either a compromised trusted setup or a falsified hardness assumption (t -SDH). The hedging technique is efficient and incurs only $O(n)$ on-chain cost to detect such failures. When detected, CHURP switches to Exp-CHURP-B that only relies on DL and no trusted setup.

As noted above, the on-chain / off-chain communication complexity of CHURP is $O(n)$ / $O(n^2)$ in the optimistic case. Unlike the optimistic path, the two pessimistic paths do not use the off-chain channel and incur $O(n^2)$ on-chain cost. Opt-CHURP and Exp-CHURP-A requires $t < n/2$, while Exp-CHURP-B requires $t < n/3$. We give more details on all the paths in CHURP in Section 4.5.

4.4 Efficient Bivariate 0-Sharing

In this section, we introduce our technique for efficient 0-sharing of bivariate polynomials. It is a key new building block in CHURP, used in the proactivation phase. The bivariate 0-sharing protocol uses resharing techniques [100, 120] as a building block.

Recall that in the context of bivariate polynomials, 0-sharing means having a committee \mathcal{C} generate a $\langle t, 2t \rangle$ -bivariate polynomial $Q(x, y)$ such that $Q(0, 0) = 0$. Each node \mathcal{C}_i holds a share $Q(i, y)$.

Previous works have naïvely extended 0-sharing techniques for univariate polynomials to the bivariate case: Each node generates its own 0-hole bivariate polynomial Q_i i.e., $Q_i(0, 0) = 0$, and distributes points on it. Thus each node transmits $O(n)$ univariate polynomials, resulting in $O(n^2)$ off-chain communication complexity per node, and $O(n^3)$ in total.

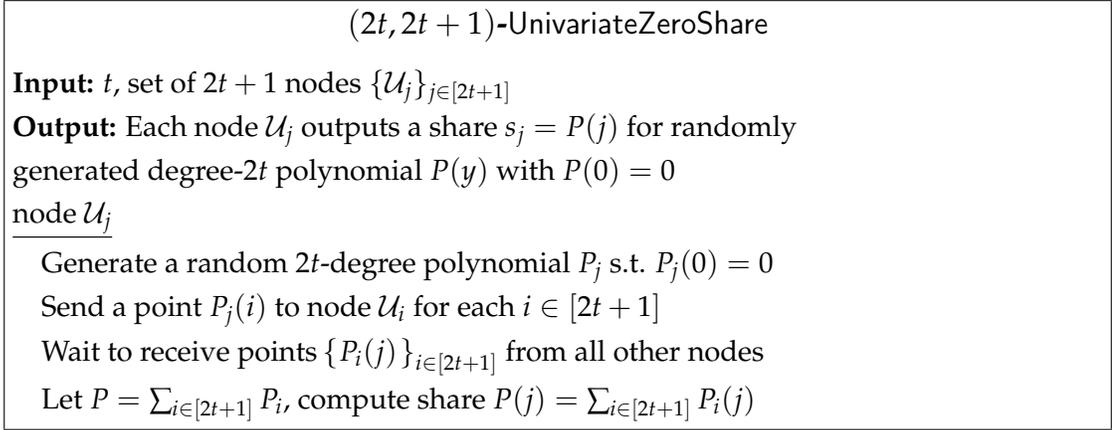


Figure 4.4: $(2t, 2t + 1)$ -UnivariateZeroShare between $2t + 1$ nodes. A 0-hole univariate polynomial P of degree- $2t$ is generated.

Our new technique, specified as protocol BivariateZeroShare, brings the total off-chain communication complexity down to just $O(tn)$ in the optimistic case. In the pessimistic case, i.e., if a node is caught cheating, different protocols (see Section 4.5) must then be invoked. Even in the pessimistic case, though, our techniques incur no more cost than in previous schemes: $O(n^3)$ in the dynamic setting and $O(n^2)$ in the static Herzberg et al. setting.

BivariateZeroShare comprises two steps. In the first step, a 0-sharing subprotocol UnivariateZeroShare is executed among a subset \mathcal{U} of $2t + 1$ nodes. At the end of this step, each node \mathcal{U}_j holds a share s_j of a univariate polynomial $P(x)$. In the second step, each node in \mathcal{U} reshapes its share s_j among all nodes, i.e., the full committee. Each node \mathcal{C}_i thereby obtains share $Q(i, y)$ of bivariate polynomial $Q(x, y)$, as desired.

BivariateZeroShare is formally specified in Fig. 4.5. (For the interest of space, we present all protocols formally in the appendix. Nonetheless, the text description here is sufficient to understand the paper.) For ease of presentation, we describe an honest-but-curious protocol version in this section. Our full protocol, which

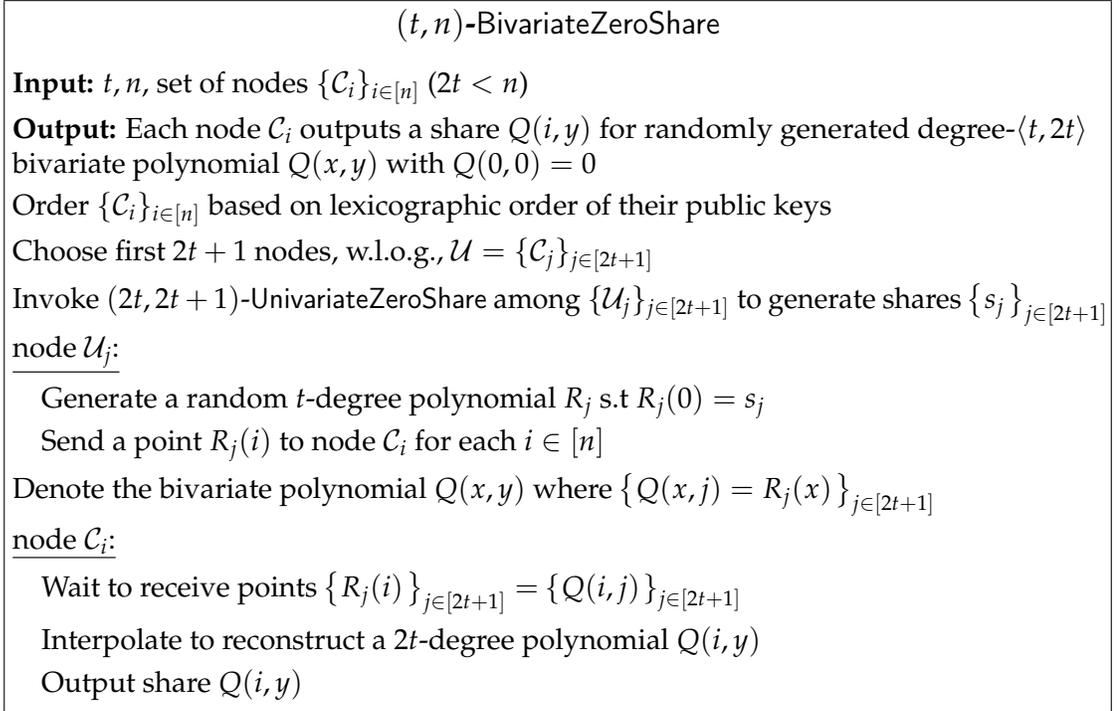


Figure 4.5: (t, n) -BivariateZeroShare between n nodes. A 0-hole bivariate polynomial Q of degree- $\langle t, 2t \rangle$ is generated.

is secure against active adversaries, is detailed in Section 4.5.3.

First step—Sharing $P(x)$: As noted, BivariateZeroShare first chooses a subset $\mathcal{U} \subseteq \mathcal{C}$ of $2t + 1$ nodes, i.e., $|\mathcal{U}| = 2t + 1$. This can be done as follows: Order nodes lexicographically by their public keys and choose the first $2t + 1$. Without loss of generality, $\mathcal{U} = \{C_j\}_{j=1}^{2t+1}$.

The nodes of \mathcal{U} then execute the univariate 0-sharing subprotocol UnivariateZeroShare presented in Fig. 4.4. This subprotocol is not new—it was previously used for proactivization in [145]. Each node \mathcal{U}_j generates a degree- $2t$ univariate 0-hole polynomial $P_j(x)$.¹ The sum $P(x) = \sum_{j=1}^{2t+1} P_j(x)$ is itself a degree- $2t$ univariate

¹An attack is outlined in [192] that breaks the UnivariateZeroShare protocol in [145]. It does so in an adversarial model similar to ours, i.e., the adversary controls t nodes in old and new committees and thus $2t$ in total, rather than t in total as in [145]. CHURP defeats this attack via dimension-switching, using reduced shares during the handoff.

0-hole polynomial $P(x)$. Then, \mathcal{U}_j redistributes points on its local polynomial $P_j(x)$, enabling every \mathcal{U}_i at the end of the step to compute its share $s_i = P(i)$.

Second step—Resharing $P(x)$: Nodes in \mathcal{U} now reshare $P(x)$ among all of \mathcal{C} , resulting in a sharing of the desired bivariate polynomial $Q(x, y)$.

Each node \mathcal{U}_j generates a degree- t univariate polynomial $R_j(x)$ uniformly at random under the constraint $R_j(0) = s_j$, i.e., $R_j(x)$ encodes the node’s share s_j . Together, the $2t + 1$ degree- t polynomials $\{R_j(x)\}$ uniquely define a degree- $\langle t, 2t \rangle$ bivariate polynomial $Q(x, y)$ such that $Q(x, j) = R_j(x)$ for $j = 1, 2, \dots, 2t + 1$ and $Q(0, 0) = 0$.

Node \mathcal{U}_j sends $R_j(i) = Q(i, j)$ to every other node \mathcal{C}_i in the full committee. Using the received points, each committee member \mathcal{C}_i interpolates to compute its share—a $2t$ -degree polynomial $Q(i, y)$. The constraint $Q(0, 0) = 0$ is satisfied because the zero coefficients of $R_j(x)$ are composed of shares generated from the 0-sharing step before, i.e., `UnivariateZeroShare`. Since each node in \mathcal{U} transmits n points, the overall cost incurred is just $O(tn)$ off-chain.

We use `(t, n)-BivariateZeroShare` as a subroutine in CHURP with some modifications. As explained before, it can also reduce the off-chain communication complexity of Herzberg et al.’s PSS scheme [145], i.e., the static-committee setting, by a factor of $O(n)$. Due to lack of space, we present this application in Appendix C.3.

4.5 CHURP Protocol Details

CHURP consists of a suite of tiered protocols with different trust assumptions

and communication complexity.

The execution starts at the top tier—a highly efficient optimistic protocol. Only upon detection of adversarial misbehavior, does the execution fall back to lower tiers. The three tiers of CHURP and their relationship are shown in Fig. 4.6, detailed as below.

The top tier, Opt-CHURP, is the default protocol of CHURP. It is optimistic and highly efficient: if no node misbehaves, the execution completes incurring only $O(n)$ on-chain and $O(n^2)$ off-chain cost. As a design choice, Opt-CHURP does not identify faulty nodes but rather just detects faulty behavior, upon which the execution switches to a lower tier protocol, also referred to as a pessimistic path.

The second tier is Exp-CHURP-A, the main pessimistic path of CHURP. Unlike the optimistic path, Exp-CHURP-A exclusively uses on-chain communication channel, which allows to identify and expel faulty nodes using proofs of correctness. Exp-CHURP-A trades performance for robustness: the execution is guaranteed to complete as long as the adversarial threshold $t < n/2$, but incurs $O(n^2)$ on-chain communication in the worst case.

Both Opt-CHURP and Exp-CHURP-A use KZG commitments to achieve $t < n/2$. As noted before, this commitment scheme requires a trusted setup phase to generate public keys with a trapdoor. The trapdoor must be “destroyed” after the setup; otherwise soundness is lost, i.e., binding property of KZG is broken. KZG introduces the only trusted setup in CHURP, and thus represents its main protocol-level vulnerability. KZG also relies on a non-standard hardness assumption, the t -Strong Diffie-Hellman assumption (t -SDH).

To hedge against soundness failure in KZG (either due to a falsified trust assumption or a compromised trusted setup), we introduce an additional verification step (StateVerif), which can be executed at the end of Opt-CHURP or Exp-CHURP-A. StateVerif is highly efficient—incur only $O(n)$ on-chain complexity. Any fault detected by StateVerif indicates that KZG is unusable, and triggers a KZG-free pessimistic path named Exp-CHURP-B. Exp-CHURP-B has the same cost as Exp-CHURP-A, but one drawback: It tolerates a lower adversarial threshold, $t < n/3$. More details on StateVerif in Section 4.5.5.

In summary, the three tiers (subprotocols) of CHURP are:

1. Opt-CHURP: The default protocol of CHURP. It incurs $O(n)$ on-chain and $O(n^2)$ off-chain communication complexity under the optimal resilience bound $t < n/2$.
2. Exp-CHURP-A: Invoked if Opt-CHURP fails. It incurs $O(n^2)$ on-chain communication complexity under the optimal bound $t < n/2$.
3. Exp-CHURP-B: Invoked if a soundness breach of KZG is detected by StateVerif. It incurs the same cost as Exp-CHURP-A, but requires $t < n/3$.

Table 4.2 summarizes the three tiers. We present only Opt-CHURP here and defer the specifications of Exp-CHURP-A and Exp-CHURP-B to Appendix C.2.

4.5.1 Notation and Invariants

We now introduce the notation and invariants that will be used to explain the protocols of CHURP. Notation is summarized in Table 4.1.

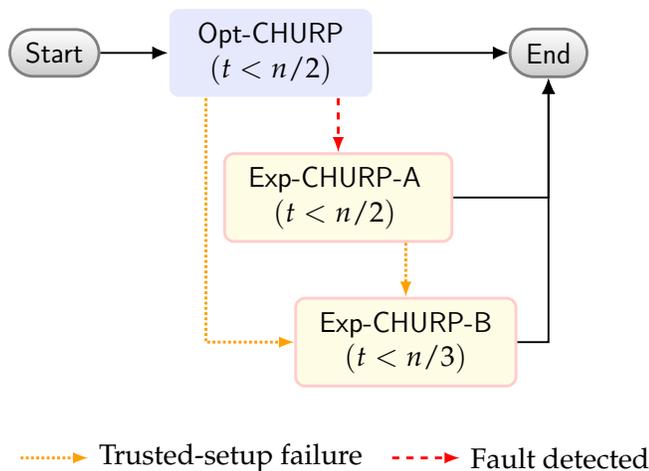


Figure 4.6: CHURP protocol tiers. Opt-CHURP is the default protocol of CHURP. Exp-CHURP-A and Exp-CHURP-B are run only if a fault occurs in Opt-CHURP.

KZG polynomial commitments: KZG commitment allows a prover to commit to a polynomial $P(x)$ and later prove the correct evaluation $P(i)$ to a verifier. (Further details in Fig. C.1 and [162].)

CHURP invariants: We say the system arrives at a *steady state* after it completes a successful handoff. The following invariants stipulate the desired properties of a steady state. We use invariants to explain the protocol and reason about its security.

Let \mathcal{C} be a committee of n nodes $\{\mathcal{C}_i\}_{i=1}^n$. Let $B(x, y)$ denote the asymmetric bivariate polynomial of degree $\langle t, 2t \rangle$ used to share the secret s , i.e., $s = B(0, 0)$. In a steady state, the three invariants below must hold:

- Inv-Secret: The secret s is the same across handoffs.
- Inv-State: Each node \mathcal{C}_i holds a full share $B(i, y)$ and a proof to the correctness thereof. Specifically, the full share $B(i, y)$ is a degree- $2t$ polynomial, and hence can be uniquely represented by $2t + 1$ points $\{B(i, j)\}_{j=1}^{2t+1}$. The proof is a set

of witnesses $\{W_{B(i,j)}\}_{j=1}^{2t+1}$.

- Inv-Comm: KZG commitments to reduced shares ($\{B(x, j)\}_{j=1}^{2t+1}$) are available to all nodes.

The first invariant Inv-Secret ensures the secret remains unchanged, a core functionality of CHURP.

Inv-State and Inv-Comm ensures the correctness of the protocol. For example, recall from Section 4.3 that during the handoff (the Share Reduction phase), nodes in the old and the new committee switch their dimension of sharing, from full shares to reduced shares. Using the commitments (specified by Inv-Comm) and the witnesses (specified by Inv-State), new committee nodes can verify the correctness of reduced shares, thus the correctness of dimension-switching.

Note that to realize Inv-Comm, hashes of KZG commitments are put on-chain for consensus while the commitments are transmitted off-chain between nodes.

| Notation | Description |
|--|--|
| $\mathcal{C}^{(e-1)}, \mathcal{C}^{(e)}$ | Old, New committee |
| $B(x, y)$ | Bivariate polynomial used to share the secret |
| $\langle t, k \rangle$ | Degree of $\langle x, y \rangle$ terms in B |
| $RS_i(x) = B(x, i)$ | Reduced share held by \mathcal{C}_i |
| $FS_i(y) = B(i, y)$ | Full share held by \mathcal{C}_i 's |
| $C_{B(x,j)}$ | KZG commitment to $B(x, j)$ |
| $W_{B(i,j)}$ | Witness to evaluation of $B(x, j)$ at i |
| $Q(x, y)$ | Bivariate proactivization polynomial |
| \mathcal{U}' | Subset of nodes chosen to participate in handoff |
| λ_i | Lagrange coefficients |

Table 4.1: Notation used in CHURP.

4.5.2 CHURP Setup

The setup phase of CHURP sets the system to a proper initial steady state. To start, an initial committee $\mathcal{C}^{(0)}$ is selected. The setup of KZG is performed and the secret is shared among $\mathcal{C}^{(0)}$. Using their shares, members of $\mathcal{C}^{(0)}$ can generate commitments to install the three invariants.

The setup of KZG can be performed by a trusted party or a committee assuming at least one of them is honest. The secret to be managed by CHURP can be generated by a trusted party or in a distributed fashion, e.g., [130]. We leave committee selection out-of-scope for this paper. Readers can refer to, e.g., [133], for a discussion.

4.5.3 CHURP Optimistic Path (Opt-CHURP)

Recall that Opt-CHURP transfers shares of some secret s from an old committee, denoted $\mathcal{C} = \mathcal{C}^{(e-1)}$, to a new committee $\mathcal{C}' = \mathcal{C}^{(e)}$. CHURP can support both committee-size and threshold changes, i.e., a transition from (n, t) to some (n', t') in any epoch. For ease of exposition here, though, we allow n to change across epochs assuming a constant threshold t . Changing the threshold is discussed in Section 4.5.4.

Opt-CHURP proceeds in three phases. The first phase, Opt-ShareReduce, performs dimension-switching to tolerate an adversary capable of compromising $2t$ nodes across the old and new committees. By the end of this phase, reduced shares are constructed by members of the new committee. The second phase, Opt-Proactivize, proactivizes these reduced shares so that new shares are indepen-

dent of the old ones. The third and the final phase, Opt-ShareDist, restores full shares from reduced shares, and thus returns to the steady state.

At the beginning of Opt-CHURP, each node in \mathcal{C}' requests the set of KZG commitments from any node in \mathcal{C} , say \mathcal{C}_1 . Recall that by the invariant Inv-Comm, each node in \mathcal{C} holds the KZG commitments to the current reduced shares, $\{C_{B(x,j)}\}_{j=1}^{2t+1}$, while the corresponding hashes are on-chain. The received commitments are verified using the on-chain hashes. Optimistically, each node in \mathcal{C}' receives the correct set of commitments. If a node receives corrupt ones, we switch to a pessimistic path where the KZG commitments are published on-chain. The above check enabled by the on-chain hashes ensures that new committee nodes receive the correct set of commitments. The phases of Opt-CHURP are as follows:

4.5.3.1 Share Reduction (Opt-ShareReduce)

The protocol starts by choosing a subset $\mathcal{U}' \subseteq \mathcal{C}'$ of $2t + 1$ members (possible because $|\mathcal{C}'| > 2t$). The nodes in \mathcal{U}' are denoted $\{\mathcal{U}'_j\}_{j=1}^{2t+1}$.

Some members in the old committee \mathcal{C} may have left the protocol by this point. Let $\mathcal{C}_{alive} \subseteq \mathcal{C}$ denote the subset of nodes that are present, w.l.o.g., let this subset be $\{\mathcal{C}_i\}_{i=1}^{|\mathcal{C}_{alive}|}$.

Recall that by the invariant Inv-State, each node \mathcal{C}_i holds a full share $B(i, y)$. Now, \mathcal{C}_i distributes points on its full share allowing computation of reduced shares $B(x, j)$ by all members of \mathcal{U}' —making a *dimension-switch* from the degree- t dimension of $B(x, y)$ to the degree- $2t$ dimension. Specifically, \mathcal{C}_i sends $B(i, j)$ to

\mathcal{U}'_j , which interpolates the received points to get its reduced share $B(x, j)$.² Note that in the optimistic path we require all $2t + 1$ nodes in \mathcal{U}' to participate. If any adversarial nodes fail to do so, we switch to a pessimistic path as detailed above.

The received points are accompanied by witnesses allowing for verification using the KZG commitments received previously. Since $t + 1$ correct points are sufficient to reconstruct the reduced share, we need at least $2t + 1$ points ($|\mathcal{C}_{alive}| > 2t$) to guarantee liveness.

The size of \mathcal{C}_{alive} is governed by the bounded churn rate α , i.e., $|\mathcal{C}_{alive}| \geq |\mathcal{C}|(1 - \alpha)$. Thus, the condition for liveness, $|\mathcal{C}_{alive}| > 2t$, places a lower bound on the committee size, $|\mathcal{C}|(1 - \alpha) > 2t$ or $|\mathcal{C}| > \lfloor 2t/1-\alpha \rfloor$.

The protocol Opt-ShareReduce is formally specified in Fig. 4.7. At the end of Opt-ShareReduce, dimension-switching is complete and each node \mathcal{U}'_j has a reduced share $B(x, j)$.

Communication complexity: Each node in \mathcal{U}' receives $O(n)$ points, so Opt-ShareReduce incurs $O(nt)$ off-chain cost.

4.5.3.2 Proactivization (Opt-Proactivize)

In this phase, \mathcal{U}' proactivizes the bivariate polynomial $B(x, y)$ —a key step in generating new shares independent of the old ones held by members of \mathcal{C} . The polynomial $B(x, y)$ is updated using a random bivariate polynomial $Q(x, y)$ generated such that $Q(0, 0) = 0$. The result is a new polynomial $B'(x, y) = B(x, y) + Q(x, y)$. The fact that $Q(0, 0) = 0$ ensures preservation of our first

²Dimension-switch can be thought as a resharing of the shares. The zero points on full shares $B(i, 0)$ i.e., shares of the secret s , are reshared.

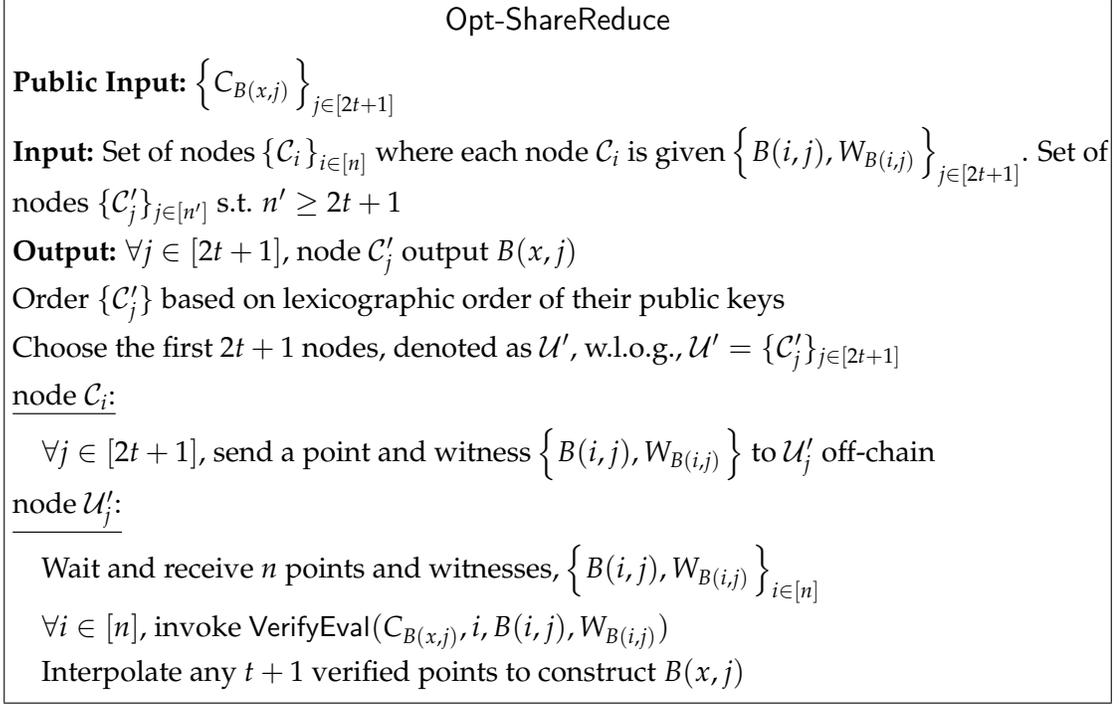


Figure 4.7: Opt-ShareReduce between the committees \mathcal{C} and \mathcal{C}' .

invariant Inv-Secret.

We achieve this by adapting the bivariate 0-sharing technique (BivariateZeroShare) presented in Section 4.4 to handle active adversaries. Recall that BivariateZeroShare comprises two steps. First, a univariate 0-sharing subroutine generates shares of the number 0. These shares are then re-shared in a second step resulting in a sharing of $Q(x, y)$ among \mathcal{C}' .

By the end of the previous, i.e., Share Reduction phase, every node \mathcal{U}'_j in the set of $2t + 1$ nodes \mathcal{U}' holds a reduced share $B(x, j)$. Now, by the end of the current, i.e., Proactivation phase, we update these reduced shares by adding $Q(x, j)$ from the generated bivariate polynomial $Q(x, y)$.

The protocol starts by invoking the 0-sharing subroutine UnivariateZeroShare introduced previously, which is the first step of BivariateZeroShare. Specifically,

$(2t, 2t + 1)$ -UnivariateZeroShare is run among \mathcal{U}' to generate shares s_j at each \mathcal{U}'_j . To handle active adversaries, \mathcal{U}'_j sends a commitment to the share, g^{s_j} , to all other nodes in \mathcal{U}' (where g is a publicly known generator). Lagrange coefficients $\{\lambda_j^{2t}\}_j$ can be precomputed to interpolate and verify if the shares form a 0-sharing, $\sum_{j=1}^{2t+1} \lambda_j^{2t} s_j = 0$. Translating it to the commitments, all nodes check the following:

$$\prod_{j=1}^{2t+1} (g^{s_j})^{\lambda_j^{2t}} = 1 . \quad (4.1)$$

Then, \mathcal{U}'_j generates a random degree- t univariate polynomial $R_j(x)$ that encodes the node's share s_j , i.e., $R_j(0) = s_j$. Together, the $2t + 1$ polynomials uniquely define a 0-hole bivariate polynomial $Q(x, y)$ such that $\{Q(x, j) = R_j(x)\}_{j=1}^{2t+1}$. \mathcal{U}'_j also updates the reduced share, $B'(x, j) = B(x, j) + R_j(x)$. Points on $B'(x, j)$ will be distributed to the entire committee \mathcal{C}' in the next phase of Opt-CHURP. (We make a modification to BivariateZeroShare: In the re-sharing step of BivariateZeroShare, points on $Q(x, j)$ were distributed directly.)

Each \mathcal{U}'_j sends constant-size information to other nodes off-chain enabling verification of the above step. Let $Z_j(x) = R_j(x) - s_j$ denote a 0-hole polynomial, the commitment to $Z_j(x)$, C_{Z_j} , and a witness to the evaluation at zero are distributed enabling verification of the statement: $Z_j(0) = 0$; equivalent to $R_j(0) = s_j$. The commitment to the updated reduced share $B'(x, j)$ is also distributed. Since $B'(x, j) = B(x, j) + Z_j + s_j$, the homomorphic property of the commitment scheme allows other nodes to verify if $C_{B'(x,j)} = C_{B(x,j)} \times C_{Z_j} \times C_{s_j}$ where $C_{s_j} = g^{s_j}$ and the other two were received previously.

In total, each node \mathcal{U}'_j generates the following set of commitment and witness information during Opt-Proactive, $\{g^{s_j}, C_{Z_j}, W_{Z_j(0)}, C_{B'(x,j)}\}$. While this set is transmitted off-chain to all nodes

in the full committee \mathcal{C}' , a hash of it is published on-chain. The received commitments can then be verified using the published hash, thereby ensuring that everyone receives the same commitments. Note that the set of commitments is sent to \mathcal{C}' instead of just the subset \mathcal{U}' to preserve the invariant *Inv-Comm*, i.e., ensure that all nodes hold KZG commitments to the updated reduced shares.

The verification mechanisms used in this protocol are sufficient to detect any faulty behavior, although they do not identify which nodes are faulty. Thus, the adversary can disrupt the protocol without revealing his / her nodes. For example, it could send corrupt commitments to nodes selectively. Although the published hash reveals this, a verifiable accusation cannot be made since the commitments were sent off-chain. Another example would be a corrupt node sending points from a non-0-hole polynomial in the *UnivariateZeroShare* protocol. Again, we detect such a fault but cannot identify which nodes are faulty. So detection of a fault simply leads to a switch to the pessimistic path, *Exp-CHURP-A*. While *Exp-CHURP-A* is capable of identifying misbehaving nodes, note that we do *not* retroactively identify the faulty nodes from *Opt-CHURP*.

The protocol *Opt-Proactivize* is formally specified in Fig. 4.8. By the end of this, if no faults are detected, each \mathcal{U}'_j holds $B'(x, j)$. The invariants *Inv-Secret* and *Inv-Comm* hold as $s = B'(0, 0)$ and all of \mathcal{C}' hold the KZG commitments respectively. In the next phase, we preserve the other invariant *Inv-State*.

Communication complexity: Each node in \mathcal{U}' publishes a hash on-chain and transmits $O(t)$ data off-chain. Hence, *Opt-Proactivize* incurs $O(t)$ on-chain and $O(t^2)$ off-chain cost.

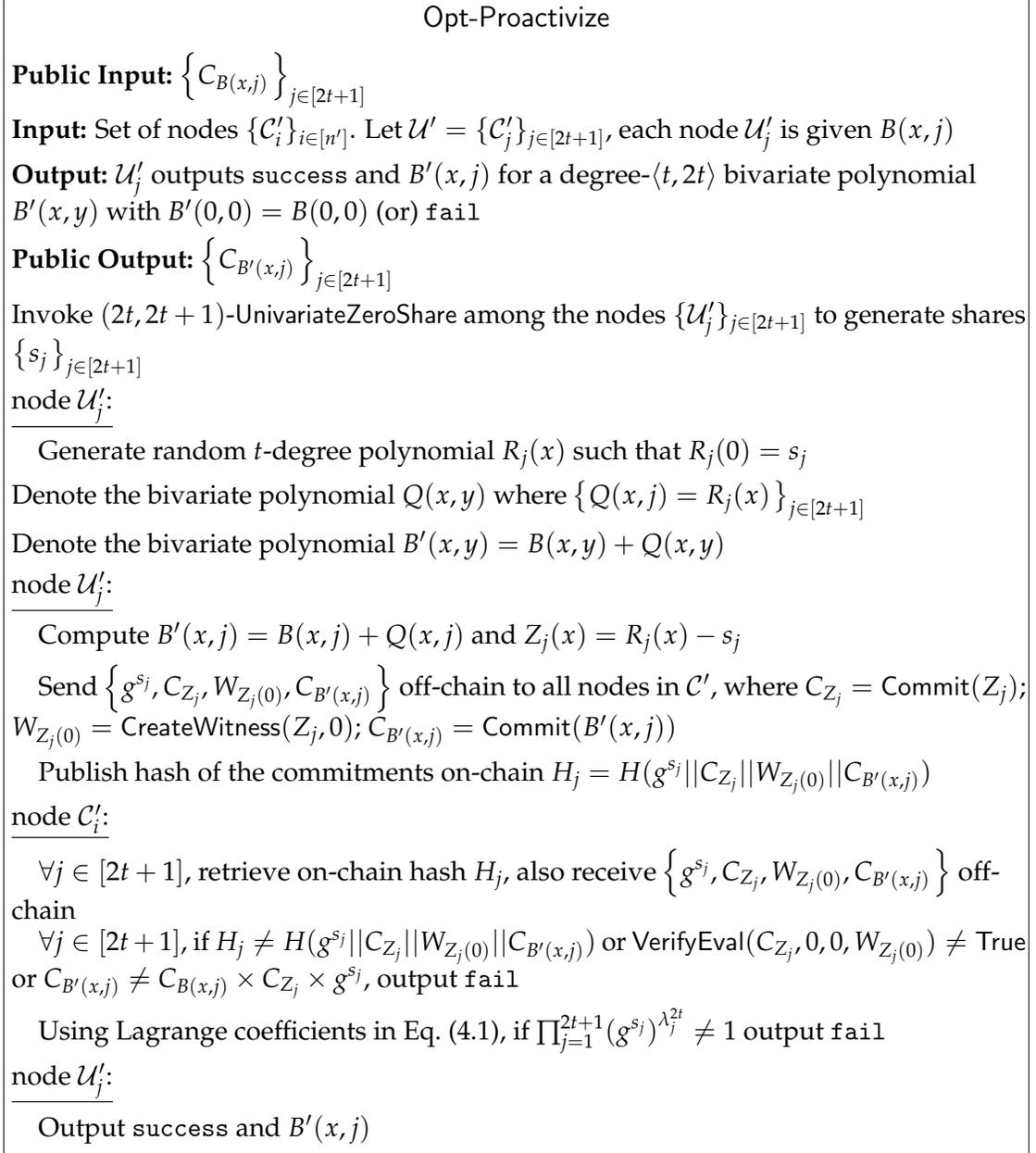


Figure 4.8: Opt-Proactivize updates the reduced shares.

4.5.3.3 Full Share Distribution (Opt-ShareDist)

In the final phase, full shares are distributed to all members of the new committee, thus preserving the Inv-State invariant. A successful completion of this phase marks the end of handoff.

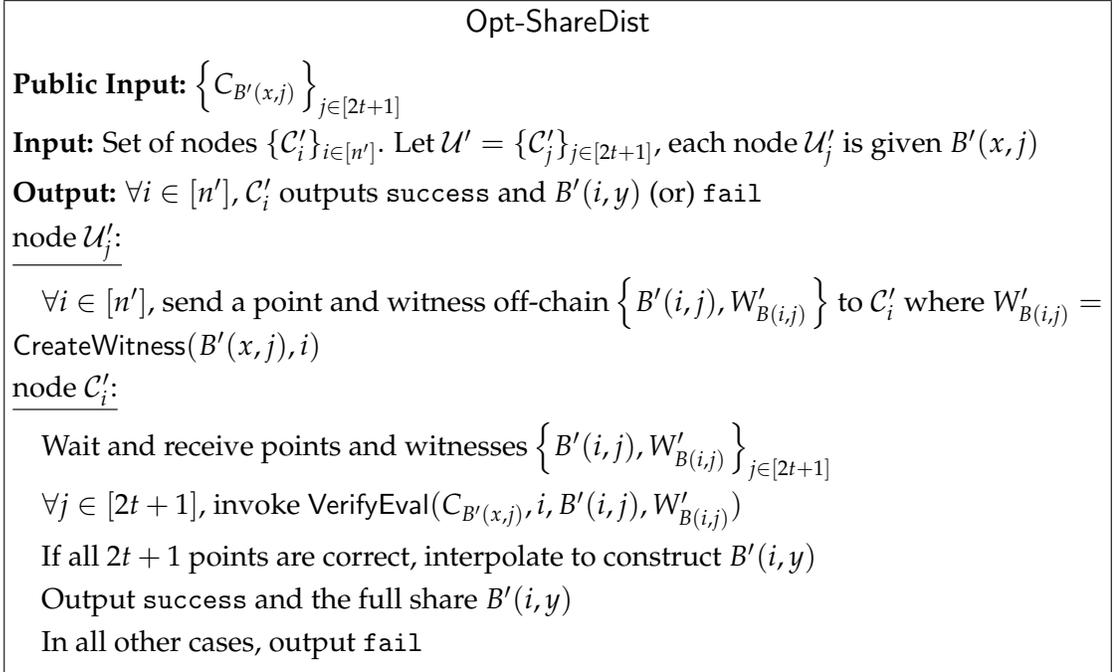


Figure 4.9: Opt-ShareDist uses the updated reduced shares to distribute full shares in C' .

By the end of the previous phase, each \mathcal{U}'_j in the chosen subset of nodes $\mathcal{U}' \subseteq C'$ holds a new reduced share $B'(x, j)$.

Now, \mathcal{U}'_j distributes points on $B'(x, j)$, allowing computation of full shares $B'(i, y)$ by all members of C' —we make a *dimension-switch* from the degree- $2t$ dimension of $B'(x, y)$ to the degree- t dimension. Specifically, each C'_i receives $2t + 1$ points $\{B'(i, j)\}_{j=1}^{2t+1}$, which can be interpolated to compute $B'(i, y)$, its full share. This is made verifiable by sending witness along with the points.

Since the point distribution is off-chain, a faulty node can send corrupt points without getting identified similar to the previous phase. In this event, we switch to the pessimistic path Exp-CHURP-A without identifying which nodes are faulty.

The protocol Opt-ShareDist is formally specified in Fig. 4.9. If all nodes receive correct points, this phase ends successfully and the optimistic path ends. The

| Protocol | On-chain, Off-chain | Threshold | Optimistic |
|------------------|---------------------|-----------|------------|
| Opt-CHURP | $O(n), O(n^2)$ | $t < n/2$ | Yes |
| Exp-CHURP-A | $O(n^2), n/a$ | $t < n/2$ | No |
| Exp-CHURP-B | $O(n^2), n/a$ | $t < n/3$ | No |
| Opt-Schultz-MPSS | $O(n), O(n^4)$ | $t < n/3$ | Yes |
| Schultz-MPSS | $O(n^2), O(n^4)$ | $t < n/3$ | No |

Table 4.2: On-chain costs and Off-chain costs for the dynamic setting. An optimistic protocol ends successfully only if no faulty behavior is detected. n/a indicates Not Applicable.

remaining invariant Inv-State is fulfilled as each node in \mathcal{C}' receives a full share, and hence the system returns to the steady state. After a successful completion of CHURP, we require that members of the old committee \mathcal{C} delete their old full shares and members of \mathcal{U}' delete their new reduced shares.

Communication complexity: Each node in \mathcal{C}' receives $2t + 1$ points, thus Opt-ShareDist incurs $O(nt)$ off-chain cost.

Each of the three phases in Opt-CHURP (and thus Opt-CHURP itself) incur no more than $O(n)$ on-chain and $O(n^2)$ off-chain cost. In terms of round complexity, it completes in three rounds (one for each phase) that does not depend on the committee size. Due to lack of space, we reiterate that the pessimistic paths of CHURP are discussed in Appendix C.2. Table 4.2 compares on-chain and off-chain costs of the three paths of CHURP and Schultz-MPSS [226], the latter will be explained in more detail in Section 4.6.3.1.

Theorem 4.1. *Protocol Opt-CHURP is a dynamic-committee proactive secret sharing scheme by Definition 4.1.*

We present the security proof in Appendix C.1.

Notes on the synchronicity assumptions As discussed in Section 4.2, CHURP works in the synchronous model and assumes a latency bound for both on-chain and off-chain communication. While the former is a well-accepted assumption (e.g., see [258, 171, 205]), the latter is assumed by the blockchain consensus protocol itself, as the required difficulty of proof-of-work is dependent on the maximum network delay [203]. However, we emphasize that synchronicity for off-chain communication is needed only for performance, *not* for liveness or safety of the full protocol. In the optimistic path, if messages take longer to deliver, a fault is detected and the protocol switches to the pessimistic path. After that, nodes communicate via the on-chain channel only.

4.5.4 Change of threshold

Thus far we have focused on schemes that allow the committee size to change while the threshold t remains constant. We now briefly describe how to enable an old committee with threshold t_{e-1} (i.e. the adversary can corrupt up to t_{e-1} nodes) to hand off shares to a new committee with a different threshold t_e .

Generally, we follow the same methodology as that of [226, 195]. To increase the threshold (i.e., $t_e > t_{e-1}$), the new committee generates a $(t_e, 2t_e)$ -degree zero-hole polynomial $Q(x, y)$ so that the proactivized sharing has threshold t_e . To reduce the threshold (i.e., $t_e < t_{e-1}$), the old committee creates $2 \times (t_{e-1} - t_e)$ *virtual servers* that participate in the handoff as honest players, but expose their shares publicly. At the end of the handoff, the new commitment incorporates the virtual servers' shares to form a sharing of threshold t_e in a similar process as the public evaluation scheme in [195].

To make changes of the threshold verifiable, we also need to extend the KZG commitment scheme with the degree verification functionality such that given a commitment $C_{\phi,d}$ to a polynomial ϕ , it can be publicly verified that ϕ is at most d -degree. Our extension relies on the q -power knowledge of exponent (q -PKE [138]) assumption. Due to lack of space, we refer readers to Appendix C.4 for more details.

4.5.5 State Verification (StateVerif)

Both Opt-CHURP and Exp-CHURP-A make use of the KZG commitment scheme, which requires a trusted setup phase and its security (binding property) relies on the t -SDH assumption. Now, we devise a *hedge* against these—a verification phase that relies only on discrete log assumptions. At a high level, StateVerif includes checks to ensure that the two important invariants, Inv-Secret and Inv-State, hold, without using the KZG commitments on-chain.

Checking Inv-Secret: Assume that the commitment to the secret g^s is on-chain from the beginning (done as part of the setup phase). Recall that at the end of Opt-CHURP or Exp-CHURP-A, each new committee node C'_i holds a full share $B'(i, y)$. The secret can also be computed from the zero points of the full shares, $s = \sum_{i=1}^n \lambda_i B'(i, 0)$, where $n = |C'|$ and $\lambda_i = \lambda_i^{n-1}$ as defined in Eq. (4.1). Each C'_i computes $s_i = B'(i, 0)$ and publishes g^{s_i} . All nodes verify that Inv-Secret remains intact by checking $g^s = \prod_{i=1}^n (g^{s_i})^{\lambda_i}$.

Checking Inv-State: In this check, we ensure that the bivariate polynomial $B'(x, y)$ is of degree $\langle t, 2t \rangle$. We achieve this by checking that the $2t + 1$ reduced shares $\{B'(x, j)\}_{j \in [2t+1]}$ are of degree t . We build an efficient procedure that

reduces the checks to a single check through a random linear combination. If the degree of $P_r(x) \stackrel{\text{def}}{=} \sum_{j=1}^{2t+1} r_j B'(x, j)$ is t , where r_j s are chosen randomly, then with high probability, the degree of all $B'(x, j)$ is t . It is important that the adversary does not know the randomness a priori, as adversarial nodes can then choose reduced shares of degree $> t$ (in the proactivization phase) in such a way that the higher degree coefficients cancel in the linear combination. In practice, r_j s can be obtained from a public source of randomness [56].

Each C'_i computes $s'_i = P_r(i) = \sum_{j=1}^{2t+1} r_j B'(i, j)$ and publishes $g^{s'_i}$ on-chain. All nodes now compute powers of the coefficients of P_r . Let $P_r(x) = \sum_{j=1}^n a_j x^j$, then $a_j = \sum_{i=1}^n \lambda_{ij} P_r(i)$, where λ_{ij} are Lagrange coefficients (an extension of Eq. (4.1)). Therefore, $g^{a_j} = \prod_{i=1}^n (g^{s'_i})^{\lambda_{ij}}$. All nodes check $\forall j > t, g^{a_j} = 1$, thus $P_r(x)$ is t -degree.

The two checks above incur $O(n)$ on-chain cost in total, thus StateVerif is highly efficient. StateVerif can fail due to two possible reasons: either the commitments are computed incorrectly by adversarial nodes, or the assumptions in the KZG scheme fail. Additional tests need to be performed to determine the cause of failure, these incur $O(n^2)$ on-chain cost and are discussed in Appendix C.2.2. If adversarial nodes are detected, the protocol expels these nodes and switches to Exp-CHURP-A. On the other hand, if KZG assumptions fail, the protocol switches to Exp-CHURP-B.

4.6 CHURP Implementation & Evaluation

We now report on an implementation and evaluation of CHURP, including a comparison with the state-of-the-art alternative, Schultz-MPSS [226].

4.6.1 Implementation

We implemented Opt-CHURP in about 2,100 lines of Go. Our implementation uses the GNU Multiprecision Library [10] and the Pairing-Based Cryptography Library [17] for cryptographic primitives, and gRPC [11] for network infrastructure.

For polynomial arithmetic, we used the polynomial ring $\mathbb{F}_p[x]$ for a 256-bit prime p . For the KZG commitment scheme, we used a type A pairing on an elliptic curve $y^2 = x^3 + x$ over \mathbb{F}_q for a 512-bit q . The order of the EC group is also p . We use SHA256 for hashing.

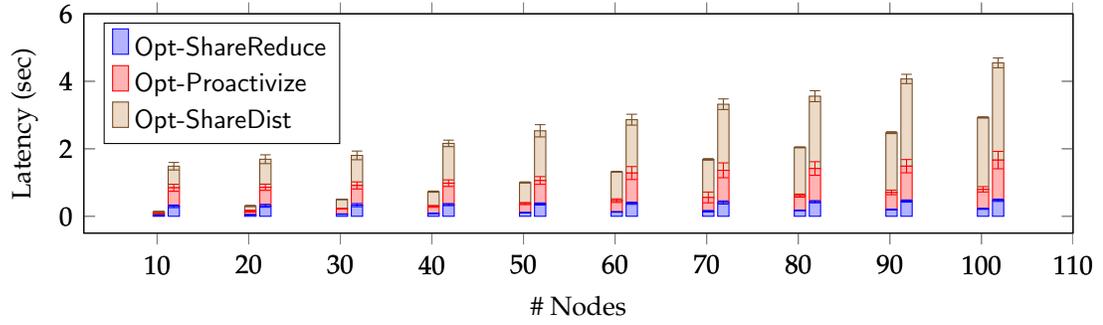
Blockchain Simulation: CHURP can be deployed on both permissioned and permissionless blockchains. We abstract away the specific choice and simulate one using a trusted node. Note that when deployed in the wild, writing to the blockchain would incur an additional constant latency.

4.6.2 Evaluation

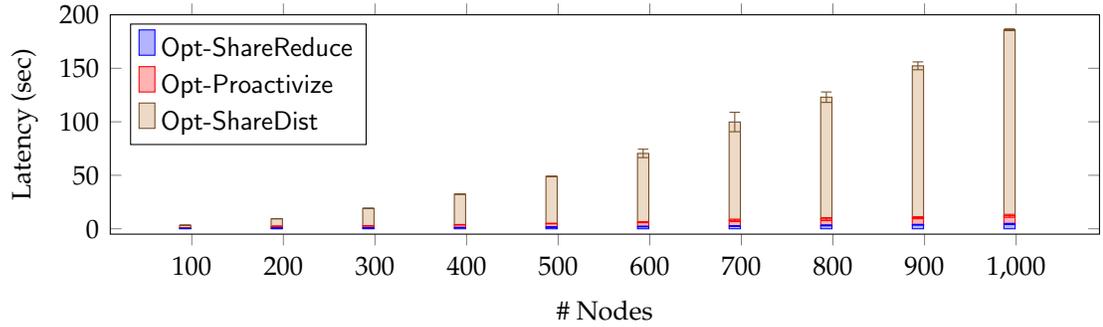
In our evaluation, experiments are run in a distributed network of up to 1000 EC2 c5.large instances, each with 2 vCPU and 4GB of memory. Each instance acts as a node in the committee and the handoff protocol is executed assuming a static committee. All experiments are averaged over 1000 epochs, i.e., 1000 invocations of Opt-CHURP. We measure three metrics for each protocol epoch: the latency (the total execution time), the on-chain complexity (the total bytes written to the blockchain (i.e. the trusted node)), and the off-chain complexity (the total bytes transmitted between all nodes). The evaluation results are presented below.

Latency: In the first set of experiments, all EC2 instances belong to the same region, also referred to as the LAN setting. This setting is useful to understand the computation time of Opt-CHURP, results are presented in Fig. 4.10. The experimental results show a quadratic increase consistent with the $O(n^2)$ asymptotic computational complexity of Opt-CHURP and suggests a low constant, e.g., for a committee of size 1001 the total protocol execution time is only about 3 minutes (Fig. 4.10b). As noted before, this does not include the additional latency for on-chain writes. Note that Opt-CHURP involves only 1 on-chain write per node which happens at the end of Opt-Proactivize, and in Ethereum currently each write takes about 15 seconds. Fig. 4.10b also shows that among the three phases, Opt-ShareDist dominates the execution time due to the relatively expensive $O(n)$ calls to KZG’s CreateWitness per node. (CreateWitness involves $O(n)$ group element exponentiation, thus total $O(n^2)$ computation.)

In the second set of experiments, we select EC2 instances across multiple regions in US, Canada, Asia and Europe, also referred to as the WAN setting. In this setting the network latency is relatively unstable, although even in the worst-case it is still sub-second. Hence, during a handoff of Opt-CHURP in the WAN setting, we expect a constant increase in the latency over the LAN setting. Moreover, we expect this constant to be relatively small compared to the time spent in computation. We validate our hypothesis—for a committee size of 100, the WAN latency is 4.54 seconds while the LAN latency is 2.92 seconds (Fig. 4.10a), i.e., the additional time spent in network latency is around 1.6 sec and constant across different committee sizes as expected. Note that we were unable to execute experiments in the WAN setting for committee sizes beyond 100 due to scaling limitations in AWS. (We plan to get around this soon.)



(a) Latency for the LAN (left bar) and WAN (right bar) setting with committee sizes 11-101.



(b) Latency for the LAN setting with committee size 101-1001.

Figure 4.10: Latency

On-chain communication complexity: Opt-CHURP incurs a linear on-chain communication complexity— n hashes, i.e. $32n$ bytes, are written to the blockchain in each handoff.

Off-chain communication complexity: Fig. 4.11 compares the off-chain complexity for different committee sizes for Opt-CHURP and [226], a discussion about the comparison is in Section 4.6.3.1. Now, we discuss the off-chain costs of Opt-CHURP. The concrete performance numbers are consistent with the expected $O(n^2)$ complexity.

The off-chain data transmitted per node includes: $2n$ (polynomial point, witness) pairs in the share reduction and the share distribution phase, and n elements of \mathbb{F}_p in the proactivization phase; each node also sends 1 commitment

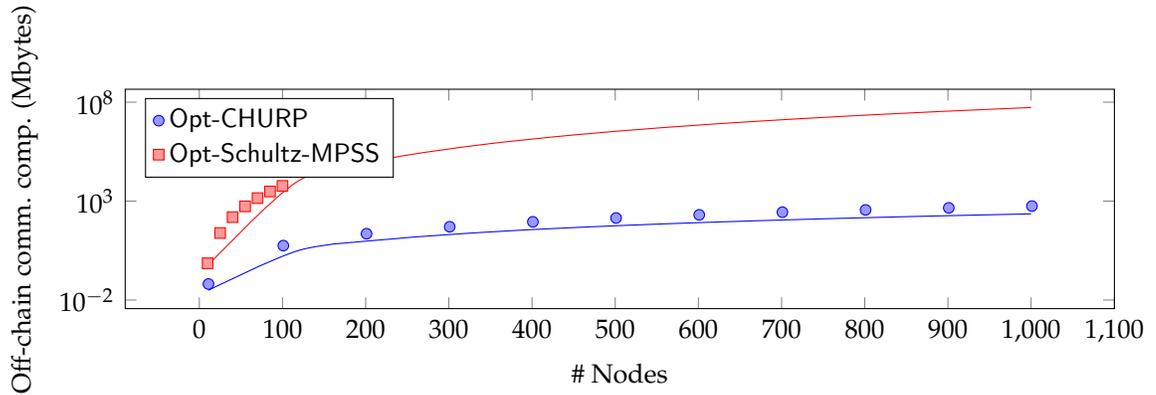


Figure 4.11: Concrete off-chain communication complexity for Opt-CHURP and Schultz-MPSS, with log-scale y-axis. Points show experimental results; expected polynomial curves (respectively quadratic and quartic) are also shown.

to share, 3 commitments to polynomials, and 1 witness. With aforementioned parameters, a commitment to a t -degree polynomial is of size 65B (with compression) and points on polynomial are of size 32B. For example, for $t = 50$ and $n = 101$, the off-chain complexity of Opt-CHURP is about $226n^2 + 325n \approx 2.3\text{MB}$. In Fig. 4.11, the expected curve is slightly below the observed data points due to trivial header messages unaccounted in the above calculations.

As we'll show now, the above is about 2300x lower than the communication complexity of the state of the art.

4.6.3 Comparison with other schemes

4.6.3.1 Schultz's MPSS

The Mobile Proactive Secret Sharing (MPSS) protocol of Schultz et al. [226], referred to as Schultz-MPSS hereafter, achieves the similar goal as CHURP in asynchronous settings, assuming $t < n/3$. Compared to [226], Opt-CHURP

achieves an $O(n^2)$ improvement for off-chain communication complexity. To evaluate the concrete performance, we also implemented the optimistic path of Schultz-MPSS (Section 5 of [226]) and evaluated the communication complexity empirically.

Asymptotic improvement: Schultz-MPSS extends the usage of expensive blinding polynomials introduced by Herzberg et al. [145] to enable a dynamic committee membership. We recall briefly the asymptotic complexity of Schultz-MPSS and refer readers to [226] for details. Each node in the old committee generates a proposal of size $O(n^2)$ and send it to other nodes, resulting in an $O(n^4)$ off-chain communication complexity in total. Each node then validates the proposals and reaches consensus on the set of proposals to use by sending $O(n)$ accusations to the primary, incurring a $O(n^2)$ on-chain communication complexity. In the optimistic case where no accusation is sent—labelled Opt-Schultz-MPSS—the consensus publishes $O(n)$ hashes of proposals on chain and thus only incurs $O(n)$ on-chain communication complexity.

Table 4.2 compares the asymptotic communication complexity of Schultz-MPSS and CHURP. Schultz-MPSS has the same on-chain complexity as CHURP, but is $O(n^2)$ more expensive for off-chain.

Performance evaluation: We implemented the optimistic path of Schultz-MPSS in about 3,100 lines of Go code. To adapt Schultz-MPSS to the blockchain setting, we replace the BFT component of Schultz-MPSS with a trusted node. Fig. 4.11 compares the off-chain communication complexity of Opt-Schultz-MPSS and Opt-CHURP.

For practical parameterizations, our experiments show that Opt-CHURP can

incur *orders of magnitude* less (off-chain) communication complexity than Opt-Schultz-MPSS. For example, for a committee of size 100, the off-chain complexity of Schultz-MPSS is $53.667n^4 \approx 5.3\text{GB}$, whereas that for Opt-CHURP is only 2.3MB, a 2300x improvement! (If $n \geq 65$, the improvement is at least three orders of magnitude.) Since Schultz-MPSS incurs excessive (GB) off-chain cost, we do not run it for committee sizes beyond 100.

4.6.3.2 Baron et al. [45]

Baron et al. devise a batched secret-sharing scheme that incurs $O(n^3)$ cost to transfer $O(n^3)$ secrets from an old to a new committee. In the single secret setting of CHURP, [45] achieves worse asymptotic cost than CHURP's optimistic path ($O(n^3)$ vs $O(n^2)$) and equivalent in the pessimistic case. The asymptotic cost, though, masks the much worse practical performance caused by the use of impractical techniques to boost corruption tolerance. The implications are twofold. First, their protocol only works when the committee size is large (hundreds to thousands as we explain below), whereas CHURP works for arbitrary committee sizes. Second, even with a large committee, their protocol requires large subgroups of nodes (hundreds to thousands) to run maliciously-secure MPC, making their protocol significantly more expensive in practice.

The bottleneck in [45] lies in the use of virtualization techniques to achieve corruption threshold close to $t < n/2$. Virtualization involves two steps: first, the committee of size n is divided into n virtual groups of size $s < n$; then each group is treated as a node in the committee to execute the protocol using MPC. [45] uses the group construction techniques of [94] that only work for large committees: for a fixed $\epsilon > 0$, to achieve a corruption threshold $t < (1/2 - \epsilon)n$, the size

| | On-chain | Transaction Ghosting |
|------------------------------------|--------------------|----------------------|
| Bandwidth (KB/sec) | ≤ 6.4 | 32.3 (9.31) |
| Latency (sec) | varies (Fig. 4.12) | 1.09 (0.82) |
| Message transmission cost (USD/MB) | varies (Fig. 4.12) | \$0.06 (\$0.02) |
| Transaction delivery rate | 100% | 92.2% (14.2%) |

Table 4.3: Comparison between communication via the Ethereum blockchain and via Transaction Ghosting. Numbers in parentheses are standard deviations. The cost for Transaction Ghosting is based on an initial gas price of 1GWei. See Section 4.7.3 for details.

of the constructed group is $16/\epsilon^2$ (See Appendix B.2 of [94]). We want ϵ to be small, e.g., $\epsilon = 0.01$ —yielding t only slightly worse than CHURP. This, however, causes the group size to explode to $s = 160,000$. Even choosing a moderate ϵ , say $\epsilon = 1/6$ —yielding $t < n/3$ which is worse than CHURP, still requires a group of size $s = 576$, meaning [45] needs to be run using maliciously-secure MPC among $n > 576$ groups of 576 nodes each, making it extremely impractical.

4.7 Point-to-Point Technique Details

CHURP takes advantage of a hybrid on-chain / off-chain communication model to minimize communication costs. A blockchain is used to reach consensus on a total ordering of messages, while much cheaper and faster off-chain P2P communication transmits messages with no ordering requirement.

Off-chain P2P channels can be implemented in different ways depending on the deployment environment. However, in a decentralized setting, establishing *direct* off-chain connection between nodes is undesirable, as it would compromise nodes’ anonymity. Revealing network-layer identities (e.g., IP addresses) would also be dangerous, as it could lead to targeted attacks. One can instead use

anonymizing overlay networks, such as Tor—but at the cost of considerable additional setup cost and engineering complexity.

Alternatively, off-chain channels can be implemented as an overlay on existing blockchain infrastructure. In this section, we present *Transaction Ghosting*, a technique for *cheap* P2P messaging on a blockchain. The key trick to reduce cost is to *overwrite transactions* so that they are broadcast, but subsequently dropped by the network. Most of these transactions—and their embedded messages—are then essentially broadcast for free. We focus on Ethereum, but similar techniques can apply to other blockchains, e.g., Bitcoin.

4.7.1 Transaction Ghosting

A (simplified) Ethereum transaction $tx = (n, m, g)$ includes a nonce n , payload m , and a per-byte *gas price* g paid to the miner of tx . For a basic (“send”) transaction, Alice pays a miner $f_0 + |m| \times g$, where f_0 is a base transaction cost and $|m|$ is the payload size. (We make this more precise below.)

Alice sends tx to network peers, who add tx to their pool of unconfirmed transactions, known as the *mempool* [210]. They propagate tx so that it can be included ultimately in all peers’ view of the mempool. tx remains in the mempool until a miner includes it in a block, at which point it is removed and $f_0 + |m| \times g$ units of currency is transferred from Alice to the miner.

The key observation is, until tx is mined, Alice can overwrite it with another transaction tx' . When this happens, tx is dropped from the mempool. Thus, both tx and tx' are propagated to all nodes, but Alice only pays for tx' , i.e., tx is

broadcast for free.

Two additional techniques can further reduce costs. Alice can embed m in tx only, putting no message data in tx' . She then only pays *nothing* for the data containing m , only the cost associated with tx' . Additionally, this technique generalizes to multiple overwrites, i.e., Alice can embed a large message m in multiple transactions $\{\text{tx}_i\}_{i \in [k-1]}$, which is useful given bounds (e.g., 32kB in Ethereum) on transaction sizes. Alice will still pay only the cost of the final transaction tx_k .

4.7.2 Choosing overwrite rate k

An optimal strategy is to make k as high as possible, i.e., overwrite many times. Ethereum, though, imposes a constraint on overwriting: the sender must raise the transaction fee in a fresh transaction by at least a minimum fraction ρ . (In Ethereum clients, ρ ranges from 10% to 12.5%).

Here we determine the optimal value of k . Recall that the fee for a transaction with $|m|$ bytes of data is $f = f_0 + g \times |m|$, for constants f_0 and g . Overwriting transactions with a fractional fee increase of ρ results in an average per-byte fee of $\frac{f \times \rho^k}{(1+\rho) \times |m|}$ for k overwritings, assuming the k th transaction gets mined. In the worst case, where $\rho = 12.5\%$, the optimal strategy is to overwrite $k = 7$ times, yielding average cost $0.29 \times \frac{f}{|m|}$ per byte, about 70% less than without overwriting.

Moreover, if we send the first $k - 1$ transactions with $|m|$ bytes of data and the last one empty, the average cost is driven down to $\frac{f_0 \times \rho^k}{|m| \times k}$ per byte (because

one only pays for the last empty transaction). As a concrete example, for $k = 7$, $|m| = 31K$, and $f = (21,000 + 68 \times m) \times 1 \text{ GWei}^3$, sending 1MB data costs about \$0.06.

The above analysis assumes the k th transaction can always successfully overwrite previous ones, which happens in our experiments for two reasons. First, the k th transaction is smaller and higher-priced, thus preferred by miners; second, previous transactions usually remain pending for a long time (tens of minutes or longer), always allowing enough time for the k th to fully propagate.

4.7.3 Experiments

We validate our ideas experimentally on the Ethereum blockchain (mainnet). The sender and receiver are full nodes connected to the Ethereum P2P network—with no out-of-band channel. The goal is for the sender to transmit messages to the receiver by embedding them in pending transactions. To overwrite a pending transaction in Ethereum, the sender reuses the same nonce and raises the gas price.

In our experiments, we rewrite $k = 7$ times. Each of the first 7 transactions contains 31KB of data and the 8th is empty. A total of approximately 100MB data is successfully transmitted in 4,200 transactions, in about 1 hour. Table 4.3 summarizes the results of our experiments, which we now discuss.

Bandwidth: DoS prevention measures and network latency in Ethereum cause overly frequent overwritten transactions to drop. Experimentally, we can propagate overwritten transactions at a rate of just under once a second, yielding

³GWei is an unit in Ethereum. 1 GWei is 10^{-9} Ether.

approximate bandwidth 32.3KB/s, as the maximum permitted per-transaction data is 32KB [83]. While this suffices for CHURP, we believed more engineering would yield higher bandwidth. Studies of blockchain arbitrage [123] show that arbitrageurs can overwrite transactions in hundreds of milliseconds.

We emphasize that the shown bandwidth is *per channel*. One can establish N concurrent channels by overwriting N transactions simultaneously.

Message-transmission cost: Transaction costs for message delivery in Transaction Ghosting are extremely low: \$0.06 per megabyte on average, with gas price 1 GWei. The gas price should be chosen minimum required to get transactions relayed by peer nodes. Empirically of late, a gas price between 1 to 2 GWei offers good delivery rate, which we now explain.

Transaction delivery rate: Although a sender can make sure overwriting succeeds in her mempool, overwritten transactions are not guaranteed to arrive on the receiver's side. Possible reasons are an overloaded mempool [210], network congestion and/or out-of-order delivery. Generally transactions with a higher transaction fee are relayed preferentially by peer nodes, and less frequently dropped. The 8th transaction in our rewriting sequence has the highest fee and the smallest payload, and is always delivered in our experiments.

Overall, we observe an average transaction delivery rate of 91.9% in our experiments, or a $\approx 9\%$ loss rate. Our Transaction Ghosting is thus an erasure channel. A sender can either erasure-code m to ensure full delivery without interaction with the receiver, or use a standard network retransmission protocol so the receiver can signal a delivery failure. These techniques are out of scope for our exploration here.

4.7.4 Comparison to on-chain communication

For comparison, we estimate the same metrics for on-chain communication, i.e. using the Ethereum blockchain as a message carrier. The results are summarized in Table 4.3.

An upper bound on the on-chain bandwidth is estimated assuming a 8 million block gas limit. Each block can hold at most three 32KB transactions, thus a total of 96KB data every 15 seconds, or 6.4 KB/s.

The message transmission cost per megabyte is estimated as that of sending 32 transactions with 32KB data in each, assuming an exchange rate of $1\text{ETH} = \$200$. The latency, i.e., the time between a transaction first appears in the mempool and the time it is mined, depends on the gas price and the network condition. A lower latency requires a higher gas price and thus a higher transmission cost. Several services such as [8, 7] collect metrics for gas price vs. latency tradeoff. We used [8] for our estimation. The tradeoff between latency and message transmission cost is shown in Fig. 4.12.

At the time of writing, gas prices in Ethereum have been consistently low for a period of approximately two months [114], preventing experimentation in a high-gas-price regime. We believe, however, that the same techniques would still work in such settings—with higher overall cost.

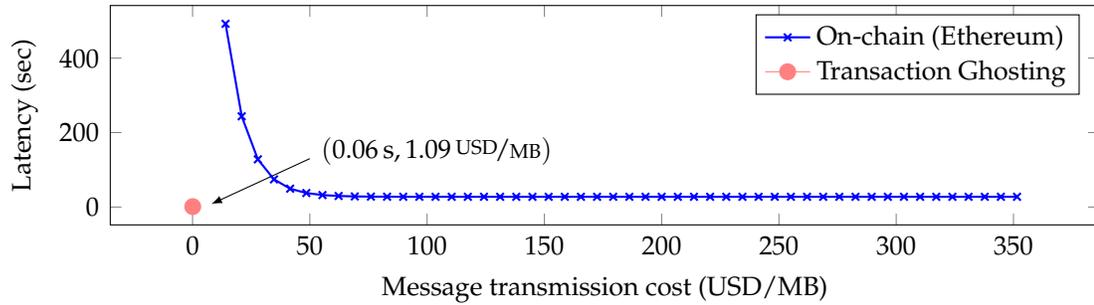


Figure 4.12: Tradeoff in latency vs. message transmission cost. The blue curve shows the observed on-chain tradeoff. The red dot at $(0.06 \text{ s}, 1.09 \text{ USD/MB})$ corresponds to Transaction Ghosting.

4.8 Applications in Decentralized Systems

Secret sharing finds use in innumerable applications involving cryptographic secrets, including secure multi-party computation (MPC) [48, 78, 89], threshold cryptography [99], Byzantine agreement [213], survivable storage systems [251], and cryptocurrency custody [36, 194], to name just a few.

Decentralized systems, however, are an especially attractive application domain, though, for two reasons.

First, blockchain systems *task individual users with management of their own private keys*, an unworkable approach for most users. A frequent result, as noted above, is key loss [217] or centralized key management [36, 194] that defeats the main purpose of blockchain systems.

Second, *blockchain objects cannot keep private state*. This fact notably limits the useful applications of smart contracts, as they cannot compute digital signatures or manage encrypted data.

We briefly enumerate a few of the most important potential applications in decentralized systems of the (dynamic-committee proactive) secret-sharing

enabled by CHURP:

Usable cryptocurrency management Rather than relying on centralized parties (e.g., exchanges) to custody private keys for cryptocurrency, or using hardware or software wallets, which are notoriously difficult to manage [31], users could instead store their private keys with committees. These committees could authenticate users and enforce access-control, resulting in the decentralized equivalent of today's exchanges.

Decentralized identity Initiatives such as the Decentralized Identity Foundation [95], which is backed by a number of major IT and services firms, as well as smaller efforts, such as uPort [26], envision an ecosystem in which users control their identities and data by means of private keys. Who will store these keys and how is left an open question [161]. The same techniques used in the cryptocurrency case for private-key management would similarly apply to assets such as identities. Additionally, a committee could manage *encrypted* identity documents on users' behalf.

Auditable access control As proposed in [169], a committee could manage encrypted documents and decrypt them for recipients under a given access-control policy while logging their accesses on-chain. The result would be a strongly auditable access-control system. This application could be managed by a smart contract.

Smart-contract attestations Committee management of smart-contract private keys could also enable *digital signing* by smart contracts. The idea would be

that committee members execute threshold signatures using a shared private key, emitting a signature for a particular smart contract in response to a request issued by the contract on chain.

Such signing would be of particular benefit in creating a simple smart-contract interface with *off-chain* systems. For example, control of Internet-of-Things (IoT) devices is commonly proposed application of smart contracts [80] (smart locks being a notable early example [209]). If smart contracts cannot generate digital signatures, then the devices they control must monitor a blockchain, an ongoing resource-intensive operation. A smart contract that can generate a digital signature, however, can simply issue authenticable commands to target devices.

Simplified Committee-based consensus for light clients A number of consensus schemes, e.g., proof-of-stake protocols [92, 65, 168, 166], aim to achieve good scalability by delegating consensus to committees. These committees change over time. Therefore verifying the blocks they sign requires awareness of their identities. By instead maintaining or only periodically rotating its public / private key pair, a committee could instead make it easier for light clients to verify signed blockchains.

Secure multiparty computation (MPC) for smart contracts More generally, dynamic-committee secret sharing would enable decentralized secure multiparty computation (MPC) by smart contracts, effectively endowing them with confidential storage and computation functionalities, as envisioned in, e.g., [263, 79].

4.9 Related Work

Verifiable Secret Sharing (VSS): Polynomial-based secret sharing was introduced by Shamir [229]. Feldman [118] and Pedersen [206] proposed an extension called *verifiable secret sharing (VSS)*, in which dealt shares’ correctness can be verified against a commitment of the underlying polynomial. In these schemes, a commitment to a degree- t polynomial has size $O(t)$. The polynomial-commitment scheme of Kate et al. [162] (KZG) reduces this to $O(1)$, and is adopted for secret sharing in, e.g., [43], and in CHURP.

KZG hedge: Prior works [143] hedge against the failure of a commitment scheme (or a cryptosystem [47]) by creating hybrid schemes that combine multiple schemes, in contrast to CHURP’s approach of using protocol tiers with different schemes in each tier. This approach coupled with novel, efficient detection techniques to switch between tiers (StateVerif), allows CHURP to include an efficient top tier (optimistic path). The notion of graceful degradation in the event of a failure appears in several works [47, 224, 125]—loosely similar to how CHURP degrades to a lower corruption threshold when the KZG scheme fails (exact notion hasn’t appeared before).

Proactive Secret Sharing (PSS): Proactive security, the idea of refreshing secrets to withstand compromise, was first proposed by Ostrovsky and Yung [201] for multi-party computation (MPC). It was first adapted for secret sharing by Herzberg et al. [145], whose techniques continue to be used in subsequent works, e.g., [129, 144, 90, 180, 226, 57, 195], and in CHURP (in UnivariateZeroShare). As noted, a result of independent interest in our work is an $O(n)$ reduction in the off-chain communication complexity of [145]. (See Appendix C.3.)

| Protocol | Dynamic | Adversary | Network | Threshold | Cost |
|-----------------------|---------|-----------|---------|-------------------------|---|
| Herzberg et al. [145] | No | active | synch. | $t < n/2$ | $O(n^2)$ |
| Cachin et al. [67] | No | active | asynch. | $t < n/3$ | $O(n^4)$ |
| Desmedt et al. [100] | Yes | passive | synch. | $t < n/2$ | $O(n^2)$ |
| Wong et al. [249] | Yes | active | synch. | $t < n/2$ | $\exp(n)$ |
| Zhou et al. [261] | Yes | active | asynch. | $t < n/3$ | $\exp(n)$ |
| Schultz-MPSS [226] | Yes | active | asynch. | $t < n/3$ | $O(n^4)$ |
| Baron et al. [45] | Yes | active | synch. | $t < n(1/2 - \epsilon)$ | $O(n^3)$ |
| CHURP (this work) | Yes | active | synch. | $t < n/2$ | $O(n^2)$ (optimistic) $O(n^3)$ (pessimistic) |

Table 4.4: Comparison of Proactive Secret Sharing (PSS) schemes—those above the line do not handle dynamic committees while the ones below do so. Cost indicates the off-chain commn. complexity.

All the above schemes assume a synchronous network model and computationally bounded adversary; CHURP does too, given its blockchain setting. PSS schemes have also been proposed in asynchronous settings [67, 261, 226] and unconditional settings [235, 196]. Nikov and Nikova [192] provide a survey of the different techniques used in PSS schemes along with some attacks (which CHURP addresses via its novel dimension-switching techniques).

Dynamic committee membership: Desmedt and Jajodia [100] propose a scheme that can change the committee and threshold in a secret-sharing system, but is unfortunately not verifiable. Wong et al. [249] build a verifiable scheme assuming that the nodes in the new committee are non-faulty. Subsequent works [261, 101, 45] build schemes that do not make such assumptions, but are impractical for our use—[261] incurs exponential communication cost, [101] incurs exponential computation cost, and [45] uses impractical virtualization techniques (See Section 4.6.3.2). Schultz et al. [226] were the *first* to build a *practical* scheme under an adversarial model similar to ours. While [226] incurs $O(n^4)$ off-chain communication cost, as Table 4.4 shows, CHURP improves it to worst-case $O(n^3)$ off-chain cost ($O(n^2)$ in the optimistic case). We convert

the on-chain cost incurred by CHURP to its equivalent off-chain cost in order to facilitate a comparison with prior work in the following manner: Instead of using a blockchain, use PBFT [74] to post messages on the bulletin board which incurs an extra $O(n)$ off-chain cost per bit.

Bivariate polynomials: Bivariate polynomials have been explored extensively in the secret-sharing literature, to build VSS protocols [67, 119], for multipartite secret-sharing [239], to achieve unconditional security [196], and to build MPC protocols [48, 131]. Prior to CHURP, few works [222, 104] have considered application of bivariate polynomials to dynamic committees, but these have been limited to passive adversaries. CHURP’s novel use of dimension-switching provides security against a strong active adversary controlling $2t$ nodes during the handoff. The dimension-switching technique applies well known resharing techniques [131, 100] via bivariate polynomials to switch between full and reduced shares.

0-sharing, the technique of generating a 0-hole polynomial has been widely used for proactive security since the work of [145]. As we explain before, prior works [222, 104, 196] have naively extended these to the bivariate case leading to expensive 0-sharing protocols. Instead, CHURP applies resharing techniques [100] to build an efficient bivariate 0-sharing protocol.

CHURP’s use of two sharings appears in some prior works [132, 212] (with largely differing goals and detail) where each node stores an additive share of the secret and a backup share of every other node’s additive share. Proactivization is achieved by resharing the additive shares, in contrast to CHURP’s approach of generating a shared polynomial explicitly which is then used to update the reduced shares. We note that adapting these techniques for use in CHURP is

non-trivial, moreover, CHURP's bivariate 0-sharing protocol has other uses as well, e.g., can reduce the off-chain cost of [145].

CHAPTER 5

REM: RESOURCE-EFFICIENT MINING FOR BLOCKCHAINS

5.1 Introduction

Despite their imperfections [73, 115, 117, 191, 221], blockchains [127, 190, 203] have attracted the interest of the financial and technology industries [41, 66, 111, 87, 208, 236] as a way to build a transaction systems with distributed trust. One fundamental impediments to the widespread adoption of decentralized or “permissionless” blockchains is that Proofs-of-Work (PoWs) in blockchains are wasteful.

PoWs are nonetheless the most robust solution today to two fundamental problems in decentralized cryptocurrency design: How to select consensus leaders and how to apportion rewards fairly among participants. A participant in a PoW system, known as a *miner*, can only lead consensus rounds in proportion to the amount of computation she invests in the system. This prevents an attacker from gaining majority power by cheaply masquerading as multiple machines. The cost, however, is the abovementioned waste. PoWs serve no useful purpose beyond consensus and incur huge monetary and environmental costs. Today the Bitcoin network uses more electricity than produced by a nuclear reactor, and is projected to consume as much as Denmark by 2020 [96].

We propose a solution to the problem of such waste in a novel block-mining system called REM. Nodes using REM replace PoW’s wasted effort with useful effort of a form that we call *Proof of Useful Work* (PoUW). In a PoUW system, users can utilize their CPUs for any desired workload, and can simultaneously

contribute their work towards securing a blockchain.

There have been several attempts to construct cryptocurrencies that recycle PoW by creating a resource useful for an external goal, but they have serious limitations. Existing schemes rely on esoteric resources [167], have low recycling rates [186], or are centralized [137]. Other consensus approaches, e.g., BFT or Proof of Stake, are in principle waste-free, but restrict consensus participation or have notable security limitations.

Intel recently introduced a new approach [87] to eliminating waste in distributed consensus protocols that relies instead on trusted hardware, specifically a new instruction set architecture extension in Intel CPUs called *Software Guard Extensions (SGX)*. SGX permits the execution of trustworthy code in an isolated, tamper-free environment, and can prove remotely that outputs represent the result of such execution. Leveraging this capability, Intel's proposed Proof of Elapsed Time (PoET) is an innovative system with an elegant and simple underlying idea. A miner runs a trustworthy piece of code that idles for a randomly determined interval of time. The miner with the first code to awake leads the consensus round and receives a reward. PoET thus promises energy-waste-free decentralized consensus with security predicated on the tamper-proof features of SGX. PoET operates in a *partially-decentralized model*, involving limited involvement of an authority (Intel), as we explain below.

Unfortunately, despite its promise, as we show in this paper, PoET presents two notable technical challenges. First, in the basic version of PoET, an attacker that can corrupt a single SGX-enabled node can win every consensus round and break the system completely. We call this the *broken chip* problem. Second, miners in PoET have a financial incentive to power mining rigs with cheap, outmoded

SGX-enabled CPUs used solely for mining. The result is exactly the waste that PoET seeks to avoid. We call this the *stale chip* problem.

REM addresses both the stale and broken chip problems. Like PoET, REM operates in a partially decentralized model: It relies on SGX to prove that miners are generating valid PoUWs. REM, however, avoids PoET’s stale chip problem by substituting PoUWs for idle CPU time, disincentivizing the use of outmoded chips for mining. Miners in a PoUW system are thus entities that use or out-source SGX CPUs for computationally intensive workloads, such as scientific experiments, pharmaceutical discovery, etc. All miners can concurrently mine for a blockchain while REM gives them the flexibility to use their CPUs for *any desired workload*.

We present a detailed financial analysis to show that PoUW successfully addresses the stale chip problem. We provide a taxonomy of different schemes, including PoW, PoET, novel PoET variants, and PoUW. We analyze these schemes in a model where agents choose how to invest capital and operational funds in mining and how much of such investment to make. We show that the PoUW in REM not only avoids the stale chip problem, but yields the smallest overall amount of mining waste. Moreover, we describe how small changes to the SGX feature set could enable even more efficient solutions.

Unlike PoET, REM addresses the broken chip problem. Otherwise, compromised SGX-enabled CPUs would allow an attacker to generate PoUWs at will, and both unfairly accrete revenue and disrupt the security of the blockchain [88, 242, 252]. Intel has sought to address the broken chip problem in PoET using a statistical-testing approach, but published details are lacking, as appears to be a rigorous analytic framework. For REM, we set forth a rig-

orous statistical testing framework for mitigating the damage of broken chips, provide analytic security bounds, and empirically assess its performance given the volatility of mining populations in real-world cryptocurrencies. Our results also apply to PoET.

A further challenge arises in REM due to the feature that miners may choose their own PoUWs workloads. It is necessary to ensure that miner-specified mining applications running in SGX accurately report their computational effort. Unfortunately SGX lacks secure access to performance counters. REM thus includes a *hierarchical attestation* mechanism that uses SGX to attest to compilation of workloads with valid instrumentation. Our techniques, which combine static and dynamic program analysis techniques, are of independent interest.

We have implemented a complete version of REM, encompassing the toolchain that instruments tasks to produce PoUWs, compliance checking code, and a REM blockchain client. As an example use, we swap REM in for the PoW in Bitcoin core. As far as we are aware, ours is the first full implementation of an SGX-backed blockchain. (Intel’s Sawtooth Lake, which includes PoET, is implemented only as a simulation.) Our implementation supports trustworthy compilation of any desired workload. As examples, we experiment with four REM workloads, including a commonly-used protein-folding application and a machine learning application. The resulting overhead is about 5 – 15%, confirming the practicality of REM’s methodology and implementation.

Paper organization

The paper is organized as follows: Section 5.2 provides background on proof-of-work and Intel SGX. We then proceed to describe the contributions of this work:

- *PoUW and REM*, a low-waste alternative to PoW that maintains PoW's security properties (Section 5.3).
- *A broken-chip countermeasure* consisting of a rigorous statistical testing framework that mitigates the impact of broken chips (Section 5.4).
- *A methodology for trustworthy performance instrumentation* of SGX applications using a combination of static and dynamic program analysis and SGX-backed trusted compilation (Section 5.5).
- *Design and full implementation of REM* as a resource-efficient PoUW mining system with automatic tools for compiling arbitrary code to a PoUW-compliant module. Ours is the first full implementation of an SGX-backed blockchain protocol (Section 5.5).
- *A model of consensus-algorithm resource consumption* that we use to compare the waste associated with various mining schemes. We overview the model and issues with previous schemes (Section 5.6), followed by a comparative application of this model to a spectrum of consensus schemes, including PoUW. (Section 5.6.1 and Section 5.6.2)

We discuss related work in Section 5.7 and conclude in Section 5.8.

5.2 Background

5.2.1 Nakamoto consensus

Blockchain protocols allow a distributed set of participants, called miners, to reach a form of consensus called Nakamoto consensus. Such consensus yields an ordered list of transactions. Roughly speaking, the process is as follows. Miners collect cryptographically signed transactions from system users. They validate the transactions' signatures and generate blocks that contain these transactions plus a pointer to a parent block. The result is a chain of blocks called (imaginatively) a blockchain.

Each miner, as it generates a block, gets to choose the block's contents, specifically which transactions will be included and in what order. System participants are connected by a peer-to-peer network that propagates transactions and blocks. Occasionally, two or more miners might nearly simultaneously generate blocks that have the same parent, forming two branches in the blockchain and breaking its single-chain structure. Thus a mechanism is used to choose which branch to extend, most simply, the longest chain available [190].¹

An attacker could naturally seek to generate blocks faster than everyone else, forming the longest chain and unilaterally choosing block contents. To prevent such an attack, a block is regarded as valid only if it contains proof that its creator has performed a certain amount of work, a proof known as a *Proof of Work* (PoW).

A PoW takes the form of a *cryptopuzzle*: In most cryptocurrencies, a miner

¹There are alternatives to this protocol [117, 175, 233, 250], however the differences are immaterial to our exploration here.

must change an input (nonce) in the block until a cryptographic hash of the block is smaller than a predetermined threshold. The security properties of hash functions force a miner to test nonces by brute force until a satisfying block is found. Such a block constitutes a solution to the cryptopuzzle and is itself a PoW. Various hash functions are used in practice. Each type puts different load on the processor and memory of a miner's computing device [190, 186, 250].

The process of mining determines an exponentially distributed interval of time between the blocks of an individual miner, and, by extension, between blocks in the blockchain. The expected amount of work to solve a cryptopuzzle, known as its *difficulty*, is set per a deterministic algorithm that seeks to enforce a static expected rate of block production by miners (e.g., 10 minute block intervals in Bitcoin). An individual miner thus generates blocks at a rate that is proportional to its *mining power*, its hashrate as a fraction of that in the entire population of miners. Compensation to miners is granted per block generated, leading to an expected miner revenue that is proportional to the miner's hashrate.

As the mining power that is invested in a cryptocurrency grows, the cryptocurrency's cryptopuzzle difficulty rises to keep the block generation rate stable. When compensation is sufficiently high, it is worthwhile for a large number of participants to mine, leading to a high difficulty requirement. This, in turn, makes it difficult for an attacker to mine a large enough fraction of blocks to perform a significant attack.

PoW properties. The necessary properties for PoW to support consensus in a blockchain, i.e., resist adversarial control, are as follows. First, a PoW must be tied to a unique block, and be valid only for that block. Otherwise, a miner can generate conflicting blocks, allowing for a variety of attacks. A PoW should

be moderately hard [38], and its difficulty should be accurately tunable so that the blockchain protocol can automatically tune the expected block intervals. Validation of PoWs, on the other hand, should be as efficient as possible, given that it is performed by the whole network. (In most cryptocurrencies today, it requires just a single hash.) It should also be possible to perform by any entity with access to the blockchain — If the proofs or data needed for validation are made selectively available by a single entity, for instance, that entity becomes a central point of control and failure.²

5.2.2 SGX

We refer readers to Section 2.2 for general background on SGX. Here we reiterate and expand on details specific to REM.

Attestation. SGX allows a remote system to verify the software running in an enclave and communicate securely with it. When an enclave is created, the CPU produces a hash of its initial state known as a *measurement*. The software in the enclave may, at a later time, request a report which includes a measurement and supplementary data provided by the process. The report is digitally signed using a hardware-protected key to produce a proof that the measured software is running in an SGX-protected enclave. This proof, known as a *quote*, is part of an *attestation* can be verified by a remote system.

SGX signs quotes in attestations using a group signature scheme called *En-*

²The Bitcoin protocol is expected to soon allow for the so-called segregated witness architecture [54, 179]. Then, transaction signatures (witnesses) are kept in a data structure that is technically separate (segregated) from the blockchain data structure. Despite this separation of data structures, the data in both must be propagated to allow for distributed validation.

hanced Privacy ID or EPID [231]. This choice of primitive is significant in our design of REM, as Intel made the design choice that attestations can only be verified by accessing Intel’s Attestation Service (IAS) [154], a public Web service maintained by Intel whose primary responsibility is to verify attestations upon request.

REM uses attestations as proofs for new blocks, so miners need to access IAS to verify blocks. The current way in which IAS works forces miners to access IAS on every single verification, adding an undesirable round-trip time to and from Intel’s server to the block verification time. This overhead, however, is not inherent, and is due only to a particular design choice by Intel. As we suggest in Section 5.5.4, a simple modification, to the IAS protocol, which Intel is currently testing, can eliminate this overhead entirely.

Randomness. As operating systems sit outside of the trusted computing base (TCB) of SGX, OS-served random functions such as `srand` and `rand` are not accessible to enclaves. SGX instead provides a hardware-protected random number generator (RNG) using the `rdrand` instruction. REM relies on the SGX RNG.

Tolerating SGX failure. SGX is known to expose some internal enclave state to the OS [252]. Our basic security model assumes ideal isolated execution, but as we detail in Section 5.4, we have baked a defense against compromised SGX CPUs into REM.

5.3 Overview of PoUW and REM

The basic idea of PoUW, and thus REM, is to replace the wasteful computation of PoW with arbitrary useful computation. A miner proves that a certain amount of useful work has been dedicated to a specific branch of the blockchain. Intuitively, due to the value of the useful work outside of the context of the blockchain supported by REM, the hardware and power are well spent, and there is no waste. A comprehensive analysis of the waste is presented in Section 5.6. Here we describe the security model of REM and then give an overview of its system mechanics.

5.3.1 Security Model

A PoW solution embodies a statistical proof of an effort spent by the miner. With PoUW, however, a miner reports its own effort. The rational miner's incentive is to lie, report more work than actually performed, and monopolize the blockchain. In PoUW / REM, use of a TEE — Intel SGX in particular — prevents such attacks and enforces correct reporting of work. The resulting trust model is starkly different from that in traditional PoW.

PoET introduced, and we similarly use in REM, a *partially decentralized* blockchain model. The blockchain is *permissionless*, i.e., any entity can participate as a miner, as in a fully decentralized blockchain such as Bitcoin. It is only partially decentralized, though, in that it relies for security on two key assumptions about the hardware manufacturer's behavior.

First, we must assume that Intel correctly manages identities, specifically that

it assigns a signing key (used for attestations) only to a valid CPU. It follows that Intel does not forge attestations and thus mining work. Such forgery, if detected in any context, would undermine the company's reputation and the perceived utility of SGX, costing far more than potential blockchain revenue. Second, we assume that Intel does not blacklist valid nodes in the network, rendering their attestations invalid when the IAS is queried. Such misbehavior would be publicly visible and similarly damaging to Intel if unjustified.

Even assuming trustworthy manufacturer behavior, though, a limited number of individual CPUs might be physically or otherwise compromised by a highly resourced adversary (or adversaries). Our trust model assumes the possibility of such an adversary and makes the strong assumption that she can learn the attestation (EPID signing) key for compromised machines and thus can issue arbitrary attestations for those machines. In particular, as we shall see, she can falsify random number generation and lie about work performed in REM.

Even this strong adversary, though, does have a key limitation: As signing keys are issued by the manufacturer, and given our first assumption above, it is not possible for an adversary to forge identities. We further assume that the signatures are linkable. In SGX, the EPID signature scheme for attestations has a linkable (pseudonymous) mode [154, 32, 231], which permits anyone to determine whether two signatures were generated by the same CPU. As a result, even a compromised node cannot masquerade as multiple nodes.

Outside the REM security model It is important to note that REM is a *consensus framework*, i.e., a means to generate blocks, not a *cryptocurrency*. REM can be integrated into a cryptocurrency, as we show by swapping it into the Bitcoin

consensus layer. As REM has roughly the same exponentially distributed block-production interval, such integration need not change security properties above the consensus layer. For example, fork resolution, transaction validation, block propagation, etc., *remain the same in a REM-backed blockchain as in a PoW-based one*. Thus we do not expand the discussion of the security issues relevant to those elements in the REM security model.

5.3.2 REM overview

Figure 5.1 presents an architectural overview of REM.

There are three types of entities in the ecosystem of REM: A *blockchain agent*, one or more *REM miners*, and one or more *useful work clients*.

The useful work clients supply useful workloads to REM miners in the form of *PoUW tasks*, each of which encompass a *PoUW enclave* and some *input*. Any SGX-compliant program can be transformed into a PoUW enclave using the toolchain we developed. Note that a PoUW enclave has to conform to certain security requirements. The most important is that it meters effort correctly, something that can be efficiently verified by a compliance checker and a novel technique we introduce called *hierarchical attestation*. We refer readers to Section 5.5.2 and Section 5.5.3 for details.

The blockchain agent collects transactions and generates a block template, a block lacking the proof of useful work (PoUW). As detailed later, a REM miner will attach the required PoUW and return it to the agent. The agent then publishes the full block to the P2P network, making it part of the blockchain and

receiving the corresponding reward.

A miner takes as input a block template and a PoUW task to produce PoUWs. It launches the PoUW enclave in SGX with the prescribed input and block template. Once the PoUW task halts, its results are returned to the useful work client. The PoUW enclave meters work performed by the miner and declares whether the mining effort is successful and results in a block. Effort is metered on a per-instruction basis. The PoUW enclave randomly determines whether the work results in a block by treating each instruction as a Bernoulli trial. Thus mining times are distributed in much the same manner as in proof-of-work systems. While in, e.g., Bitcoin, effort is measured in terms of executed hashes, in REM, it is the number of executed useful-work instructions. Intuitively, REM may be viewed as *simulating* the distribution of block-mining intervals associated with PoW, but REM does so with PoUW, and thus eliminates wasted CPU effort.

When a PoUW enclave determines that a block has been successfully mined, it produces a PoUW, which consists of two parts: an SGX-generated attestation demonstrating the PoUW enclave's *compliance* with REM and another attestation that a block was successfully mined by the PoUW enclave at a given *difficulty parameter*. The blockchain agent concatenates the PoUW to the block template, forming a full block, and publishes it to the network.

When a blockchain participant verifies a fresh block received on the blockchain network, in addition to verifying higher-layer properties (e.g., in a cryptocurrency such as Bitcoin, that transactions, previous block references, etc., are valid), the participant verifies the attestations in the associated PoUW.

Intel's PoET scheme looks similar to REM in that its enclave randomly de-

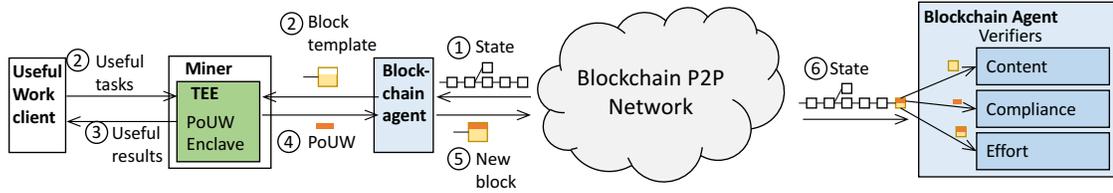


Figure 5.1: Architecture overview of REM.

terminates block intervals and attests to block production. PoET, however, lacks the production of useful work, an essential ingredient, as we explain later in the paper. We now discuss our strategy in REM for handling compromised nodes.

5.4 Tolerating Compromised SGX Nodes

SGX does not achieve perfect enclave isolation. While no real practical attack is known, researchers have demonstrated potentially dangerous side-channel attacks against applications [252] and even expressed concerns about whether an attestation key might be extracted [88].

Therefore, even if we assume SGX chips are manufactured in a secure fashion, some number of individual instances could be broken by well-resourced adversaries. A single compromised node could be catastrophic to an SGX-based cryptocurrency, allowing an adversary to create blocks at will and perform majority attacks on the blockchain. While she could not spend other people’s money, which would require access to their private keys, she could perform denial-of-service attacks, selectively drop transactions, or charge excessive transaction fees.

In principle, a broken attestation key can be revoked through the Intel Attestation Service (IAS), but this can only happen if the break is detected to begin

with. Consequently, Intel has explored ways of detecting SGX compromise in PoET [19] by statistically testing for implausibly frequent mining by a given node (using a “z-test”). Details are lacking in published materials, however, and a rigorous analytic framework seems to be needed.

For REM, we explore compromise detection within a rigorous definitional and analytic framework. The centerpiece is what we call a *block-acceptance policy*, a flexibly defined rule that determines whether a proposed block in a blockchain is legitimate. As we show, defining and analyzing policies rigorously is challenging, but we provide strong analytical and empirical evidence that a relatively simple statistical-testing policy (which we denote P_{stat}) can achieve good results. P_{stat} both limits an adversary’s ability to harvest blocks unfairly and minimizes erroneous rejection of honestly mined blocks.

5.4.1 Threat Model and Definitions

5.4.1.1 Basic notation

To model block-acceptance policies, let $M = \{m_1, \dots, m_n\}$ be the set of all miners, which we assume to be static. (Miners can join and leave the system; M includes all potential miners.) An adversary \mathcal{A} controls a static subset $M_{\mathcal{A}} \in M$, where $|M_{\mathcal{A}}| = k$. $\text{rate}(m_i)$ specifies the *mining rate* of m_i , the number of mining operations per unit time it performs.

We define a candidate *block* to be a tuple $B = (t, m, d)$, where t is a timestamp, $m \in M$ the identity of the CPU that mines the block, and d is the block difficulty. Difficulty d is defined as the win probability per mining operation in the under-

lying consensus protocol (e.g. a hash in Bitcoin, a unit time of sleep in PoET, an instruction in PoUW). \mathcal{B} denotes the set of possible blocks B .

A *blockchain* is an ordered sequence of blocks. At time τ , blockchain $C(\tau)$ is a sequence of accepted blocks $C(\tau) = \{B_1, B_2, \dots, B_n\}$ for some n . We drop τ where its clear from context. We let $r(\tau)$ denote the number of rejected blocks of honest miners, i.e., miners in $M - M_{\mathcal{A}}$, in the history of $C(\tau)$. (Of course, $r(\tau)$ is not and indeed cannot be recorded in a real blockchain system.) Let \mathcal{C} be the space of all possible blockchains C . Let C_m denote blockchain C restricted to blocks mined by miner $m \in M$.

In REM, a blockchain-acceptance policy is used to determine whether a block appears to come from a legitimate miner (CPU that hasn't been compromised).

Definition 5.1. (*Blockchain-Acceptance Policy*) A *blockchain-acceptance policy* (or *simply policy*) $P : \mathcal{C} \times \mathcal{B} \rightarrow \{\text{reject}, \text{accept}\}$ is a function that takes as input a blockchain and a proposed block, and outputs whether the proposed block is legitimate.

5.4.1.2 Security and efficiency definitions

We model the consensus algorithm for the blockchain, the adversary \mathcal{A} , and honest miners respectively as (ideal) programs $\text{prog}_{\text{chain}}$, $\text{prog}_{\mathcal{A}}$, and prog_m . Together, they define what we call a *security game* $S(P)$ for a particular policy P .

We define security games and their constituent programs formally in Appendix D.1.2. Where clear from context in what follows, we use the notation S , rather than $S(P)$, i.e., omit P .

A security game S may itself be viewed as a probabilistic algorithm. Thus we

may treat the blockchain resulting from execution of S for interval of time τ as a random variable $C_S(\tau)$.

Normalizing the revenue from mining a block to 1, we define the *payoff* for a miner m for a given blockchain C as $\pi_m(C) = |C_m|$.

An adversary \mathcal{A} seeks to maximize payoffs for its miners, as reflected in the following definition:

Definition 5.2. (*Advantage of \mathcal{A}*). For a given security game S , the advantage of \mathcal{A} for time τ is:

$$\text{Adv}_{\mathcal{A}}^S(\tau) = \frac{\mathbb{E}[\pi_{\hat{m}}(C_S(\tau))]}{\max_{m_j \in M - M_{\mathcal{A}}} \mathbb{E}[\pi_{m_j}(C_S(\tau))]},$$

for any $\hat{m} \in M_{\mathcal{A}}$. Note that $\mathbb{E}[\pi_{\hat{m}}(C_S(\tau))]$ is equal for all such \hat{m} , as they all use strategy $\Sigma_{\mathcal{A}}$ and can emit blocks as frequently as desired (ignoring $\text{rate}(\hat{m})$).

A policy that keeps $\text{Adv}_{\mathcal{A}}^S(\tau)$ low is desirable, but there's a trade-off. A policy that rejects too many policies incurs high *waste*, meaning that it rejects many blocks from honest miners. We define waste as follows.

Definition 5.3. (*Waste of a policy*). For a given blockchain $C(\tau) = \{(B_1, B_2, \dots, B_n)\}$, the waste is defined as

$$\text{Waste}(C(\tau)) = \frac{r(\tau)}{n + r(\tau)}.$$

For security game S , the waste at time τ is defined as

$$\text{Waste}^S(\tau) = \mathbb{E}[\text{Waste}(C_S(\tau))].$$

Our exploration of policies focuses critically on the trade-offs between low $\text{Adv}_{\mathcal{A}}^S(\tau)$ and low $\text{Waste}^S(\tau)$. To illustrate the issue, we give a simple example in Appendix D.1.3 of a policy that allows any CPU to mine only one block over its lifetime. As $\tau \rightarrow \infty$, it achieves the optimal $\text{Adv}_{\mathcal{A}}^S(\tau) = 1$, but at the cost of $\text{Waste}^S(\tau) = 1$, i.e., 100% waste.

5.4.2 The REM policy: P_{stat}

REM makes use of a statistical-testing-based policy that we denote by P_{stat} . P_{stat} is compatible not just with PoUW, but also with PoET and potentially other SGX-based mining variants.

There are two parts to P_{stat} . First, P_{stat} estimates the rate of the fastest honest miner(s) (fastest CPU type), denoted by $\text{rate}_{\text{best}} = \max_{m \in M - M_{\mathcal{A}}} \text{rate}(m)$. There are various ways to accomplish this; a simple one would be to have an authority (e.g., Intel) publish specs on its fastest CPUs' performance. (In PoET, mining times are uniform, so $\text{rate}_{\text{best}}$ is just a system parameter.) We describe an empirical approach to estimating $\text{rate}_{\text{best}}$ in REM in Appendix D.1.1.

Given an estimate of $\text{rate}_{\text{best}}$, P_{stat} tests submitted blocks statistically to determine *whether a miner is mining blocks too quickly and may thus be compromised*. The basic principle is simple: On receiving a block B from miner m , P_{stat} tests the null hypothesis

$$H_0 = \{\text{rate}(m) \leq \text{rate}_{\text{best}}\}.$$

We use $|C_m(\tau)|$, the number of blocks mined by m at time τ , as the test statistic.

```

 $P_{\text{stat}}^{\alpha, \text{rate}_{\text{best}}}(C, B):$ 
  parse  $B \rightarrow (\tau, m, d)$ 
  if  $|C_m| > F^{-1}(1 - \alpha, d\tau(\text{rate}_{\text{best}})):$ 
    output reject
  else
    output accept

```

Figure 5.2: P_{stat}^{α} . $F^{-1}(\cdot, \lambda)$ is the quantile function for Poisson distribution with rate λ .

Under H_0 , $|C_m|$ should obey a Poisson distribution with rate $d\tau(\text{rate}_{\text{best}})$, denoted as $\text{Pois}[d\tau(\text{rate}_{\text{best}})]$. P_{stat} rejects H_0 if $|C_m|$ is greater than the $(1 - \alpha)$ -quantile of the Poisson distribution. The false rejection rate for a single test is therefore at most α . We specify P_{stat} (for a given $\text{rate}_{\text{best}}$) in Fig. 5.2.

An important property that differentiates P_{stat} from canonical statistical tests is that P_{stat} repeatedly applies a given statistical test to an accumulating history of samples. *The statistical dependency between samples makes the analysis non-trivial, as we shall show.*

5.4.3 Analysis of P_{stat}

We now analyze the average-case and worst-case waste and adversarial advantage of P_{stat} . We assume for simplicity that $\text{rate}_{\text{best}}$ is accurately estimated. We remove this assumption in the worst-case analysis below. We also assume that the difficulty $d(t)$ is stationary over the period of observation.

Waste Under P_{stat} , a miner generates blocks according to a Poisson process; whether a block is accepted or rejected depends on whether the miner has generated more blocks than a time-dependent threshold. This process is obviously

not memoryless and thus not directly representable as a Markov process. We can, however, achieve a close approximation using a discrete-time Markov chain. Indeed, as we show, we can represent waste in P_{stat} using a discrete-time Markov chain that is *periodically identical* to the process it models, meaning that its expected waste is identical at any time $n\tau$, for $n \in \mathbb{Z}^+$ and τ a model parameter specified below. This Markov chain has a stationary distribution that yields an expression upper-bounding waste in P_{stat} . (We believe, and the periodic identical property suggests, that this bound is very tight.)

To construct the Markov Chain, we partition time into intervals of length τ ; we regard each such interval as a discrete timestep. Assuming that all honest miners mine at rate $rate$, let $\lambda = d\tau(rate)$. Thus an honest miner generates an expected $\text{Pois}[\lambda]$ blocks in a given timestep i , which we may represent as a random variable Y_i . Without loss of generality, we may set $\tau = 1/(d \times rate)$ and thus $\lambda = 1$ and $E[\text{Pois}[\lambda]] = 1$.

We represent the state of an honest miner at timestep n by a random variable $X_n = \sum_{i=1}^n (Y_i - E[Y_i]) = (\sum_{i=1}^n Y_i) - n$. Thus $X_n \in \mathbb{Z}$ is simply difference between the miner's actually mined blocks and the expected number.

Our Markov chain consists of a set of states $C = \mathbb{Z}$ representing possible values of X_n (we use the notation C here, as states represent $|C_m|$ for an honest miner m). Figure 5.3 gives a simple example of such a chain (truncated to only four states).

Our statistical testing regime may be viewed as rejecting blocks when a transition is made to a state whose value is above a certain threshold $thresh$. We denote the set of such states $C_{\text{rej}} = \{j \mid j \geq thresh\} \in C$ and depict corresponding

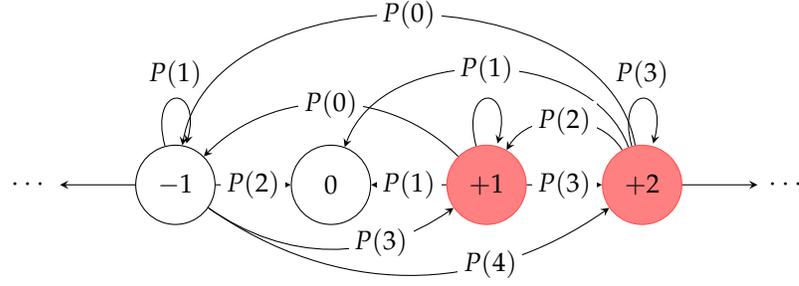


Figure 5.3: Markov chain with states C representing P_{stat} . Red nodes show the rejection set $C_{\text{rej}} = \mathbb{Z}^+$, i.e., $\text{thresh} = 1$. Outgoing edges from 0 are omitted for clarity.

nodes visually in our example in Figure 5.3 as red. P_{stat} sets thresh according to the statistical-testing regime we describe above and a desired false-rejection (Type-I) parameter α . Specifically,

$$C_{\text{rej}}[\alpha] = \{j \in \mathbb{Z} \mid j \geq F^{-1}(1 - \alpha, \tau \times \text{rate})\}. \quad (5.1)$$

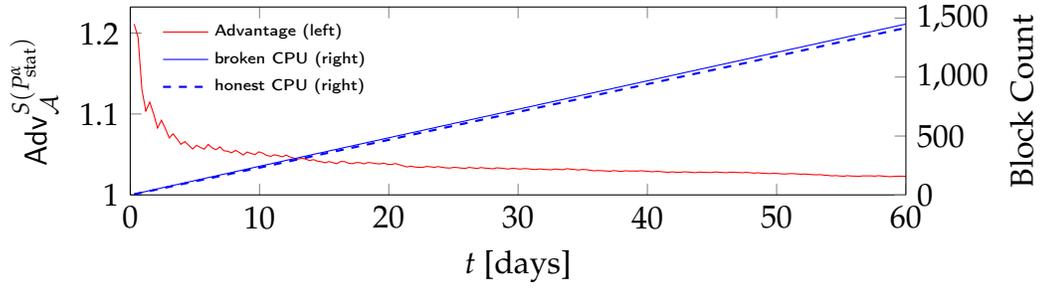
The transition probabilities in our Markov chain are:

$$P[i \rightarrow j \mid i \in C \setminus C_{\text{rej}}[\alpha]] = \begin{cases} P(j - i + 1) & \text{if } j \geq i - 1 \\ 0 & \text{otherwise} \end{cases} \quad (5.2)$$

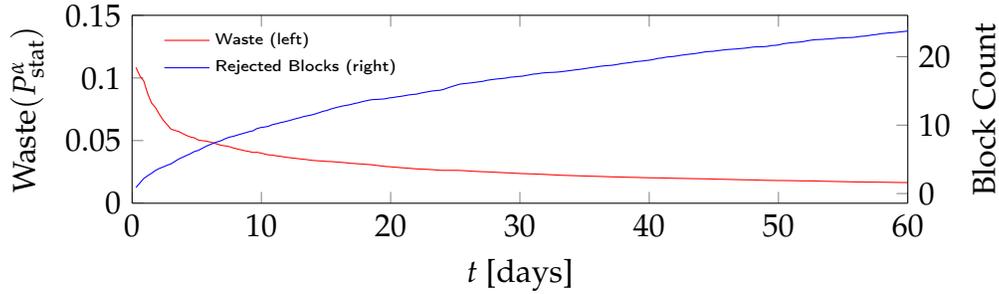
$$P[i \rightarrow j \mid i \in C_{\text{rej}}[\alpha]] = \begin{cases} P(j + 1) & \text{if } j \leq -1 \\ 0 & \text{otherwise.} \end{cases} \quad (5.3)$$

An example of transitions is given in Fig. 5.3. For instance, from state -1 , the next state can be -2 if the miner doesn't produce any block in this step with probability $P(0)$, or state $-2 + i$ if the miner produces $i + 1$ blocks in this step, thus with probability $P(i + 1)$.

Finally, an upper bound on the false rejection rate can be derived from the stationary probabilities of the Markov chain. Letting $q(s)$ denote the stationary



(a) Left y -axis: adversarial advantage of P_{stat} . Right y -axis: the number of blocks mined by a compromised CPU versus an honest CPU.



(b) Left y -axis: the waste of P_{stat} . Right y -axis: the number of rejected blocks.

Figure 5.4: 60-day simulation of P_{stat} . The fastest honest CPU mines one block per hour. The Markov chain analysis yields a long-term advantage upper bound of 1.006 and waste of 0.006.

probability of state s ,

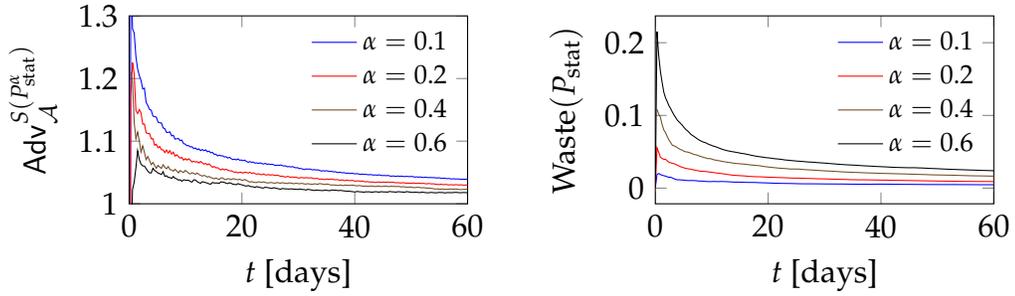
$$\text{Waste}(P_{\text{stat}}^\alpha) = \sum_{s \in C_{\text{rej}}[\alpha]} sq(s). \quad (5.4)$$

We compare our analytic bounds with simulation results in below.

Adversarial Advantage We denote by Σ_{stat} the strategy of an adversary that publishes blocks as soon as they will be accepted by P_{stat} . In Appendix D.1.4, we show the following:

Theorem 5.1. *In a (non-degenerate) security game S where \mathcal{A} uses strategy Σ_{stat} ,*

$$\text{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^\alpha)} = \frac{1}{1 - \text{Waste}(P_{\text{stat}}^\alpha)}.$$



(a) The adversarial advantage of P_{stat} under different α (b) The waste of P_{stat} under different α

Figure 5.5: 60-day simulation of P_{stat} , under various α . The fastest honest CPU mines an expected one block per hour.

Simulation We simulate P_{stat} to explore its efficacy in both the average case and the worst case. Fig. 5.4 shows the result of 1000 runs of a 60-day mining period simulation under P_{stat} . We set $\alpha = 0.4$. We present statistics with respect to the fastest (honest) CPU in the system, which for simplicity we assume mines one block per hour in expectation and refer to simply as “the honest miner.” The adversary uses attack strategy Σ_{stat} .

In Fig. 5.4a, the solid blue line shows the average aggregate number of blocks mined by the adversary, and the dashed one those of the honest miner. The attacker’s advantage is, of course, the ratio of the two values. Initially, the adversary achieves a relatively high advantage ($\approx 127\%$), but this drops below 110% within 55 blocks, and continues to drop. Our asymptotic analytic bound on waste (given below) implies an advantage of 100.6%.

Fig. 5.4b shows the average waste of P_{stat} and absolute number of rejected blocks. The waste quickly drops below 10%. As blocks accumulate, the statistical power of P_{stat} grows, and the waste drops further. Analytically, we obtain $\text{Waste}(P_{\text{stat}}^\alpha) = 0.006$, or 0.6% from Eqn. 5.4.

Setting α Setting the parameter α imposes a trade-off on system implementers. As noted, α corresponds to the Type-I error for a single test in P_{stat} . As P_{stat} performs continuous testing, however, a more meaningful security measure is $\text{Waste}(P_{\text{stat}}^\alpha)$, the rate of falsely rejected blocks. Similarly there is no notion of Type-II error—particularly, as our setting is adversarial. $\text{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^\alpha)}$ captures the corresponding notion in REM. As shown in Figure 5.5, raising α results in a lower $\text{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^\alpha)}$, but higher $\text{Waste}(P_{\text{stat}}^\alpha)$, and vice versa.

5.5 Implementation Details

We have implemented a full REM prototype using SGX (Section 5.5.1), and as an example application swapped REM into the consensus layer of Bitcoin-core [54]. We explain how we implemented secure instruction counting (Section 5.5.2), and our hierarchical attestation framework (Section 5.5.3) that allows for arbitrary tasks to be used for work. We explain how to reduce the overhead of attestation due to SGX-specific requirements (Section 5.5.4). Finally (Section 5.5.5) we present two examples of PoUW and evaluate the overhead of REM.

5.5.1 Architecture

Figure 5.1 shows the architecture of REM. As discussed in Section 5.3.2, the core of REM is a miner program that does useful work and produces PoUWs. Each CPU instruction executed in the PoUW is analogous to one hash function computation in PoW schemes. That is, each instruction has some probability of successfully mining a block, and if the enclave determines this is the case, it

```

Miner loop
while true:
    template ← read from blockchain agent
    hash, difficulty ← process(template)
    task ← get from useful work client
    outcome, PoUW ← TEE(task, hash, difficulty)
    send outcome to useful work client
    if PoUW ≠ ⊥ then
        block ← formBlock(template, PoUW)
        send block to blockchain agent
    fi
endwhile

```

Figure 5.6: Miner Loop. The green highlighted line is executed in a TEE (e.g., an SGX enclave).

produces a proof — the PoUW.

Pseudocode of the miner’s iterative algorithm is given in Algorithm 5.6. In a given iteration, it first takes a block template from the agent and calculates the previous block’s hash and difficulty. Then it reads the task to perform as useful work. Note that the enclave code has no network stack, therefore it receives its inputs from the miner untrusted code and returns its outputs to the miner untrusted code. The miner calls the TEE (SGX enclave) with the useful task and parameters for mining, and stores the result of the useful task. It also checks whether the enclave returned a successful PoUW; if so, it combines the agent-furnished template and PoUW into a legal block and sends it to the agent for publication. In REM, the miner untrusted layer is implemented as a Python script using RPC to access the agent.

To securely decide whether an instruction was a “winning” one, the PoUW enclave does the equivalent of generating a random number and checking whether

it is smaller than value `target` that represents the desired system-wide block rate, i.e., difficulty. For this purpose, it uses SGX's random number generator (SRNG). However, calling the SRNG and checking for a win after every single instruction would impose prohibitive overhead. Instead, we batch instructions by dividing useful work into subtasks of short duration compared to the inter-block interval (e.g. 10 second tasks for 10 minute average block intervals). We let each such subtask run to completion, and count its instructions. The PoUW enclave then calls the SRNG to determine whether at least one of the instructions has won, i.e., it checks for a result less than `target`, weighted by the total number of executed instructions. If so, the enclave produces an attestation that includes the input block hash and difficulty.

Why Count Instructions While instructions are reasonable estimates of the CPU effort, CPU cycles would have been a more accurate metric. However, although cycles are counted, and the counts can be accessed through the CPU's performance counters, they are vulnerable to manipulation. The operating system may set their values arbitrarily, allowing a rational operator, who controls her own OS, to improve her chances of finding a block by faking a high cycle count. Moreover, counters are incremented even if an enclave is swapped out, allowing an OS scheduler to run multiple SGX instances and having them double-count cycles. Therefore, while instruction counting is not perfect, we find it is the best method for securely evaluating effort with the existing tools available in SGX.

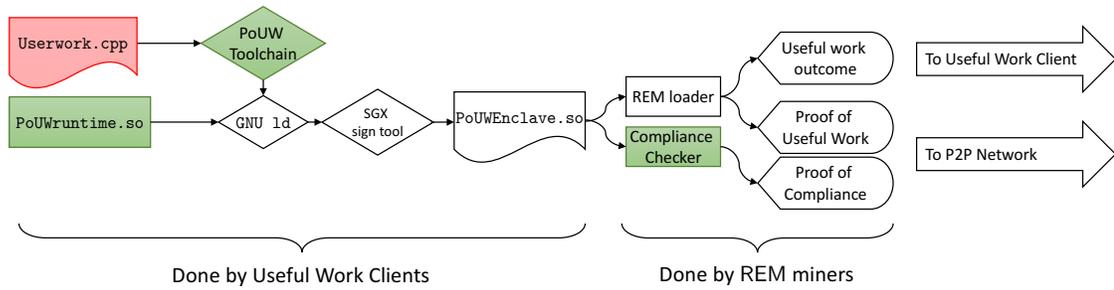


Figure 5.7: REM toolchain to transfer a useful work to an PoUW-ready program. Everything in the diagram has been implemented besides existing tools such as ld and SGX signing tool.

5.5.2 Secure Instruction Counting

As we want to allow arbitrary useful work programs, it is critical to ensure that instructions are counted correctly even in the presence of malicious useful work programs. To this end, we adopt a hybrid method combining static and dynamic program analysis. We employ a customized toolchain that can instrument any SGX-compliant code with dynamic runtime checks implementing secure instruction counting.

Figure 5.7 shows the workflow of the PoUW toolchain. First, the useful work code (`usefulwork.cpp`), C / C++ source code, is assembled while reserving a register as the instruction counter. Next, the assembly code is rewritten by the toolchain such that the counter is incremented at the beginning of each basic block (a linear code sequence with no branches) by the number of instructions in that basic block. In particular, we use the LEA instruction to perform incrementing for two reasons. First, it completes in a single cycle, and second, it doesn't change flags and therefore does not affect conditional jumps. The count is performed at the beginning of a block rather than its end to prevent a cheater from jumping to the middle of a block and gaining an excessive count.

```

PoUW Runtime
TEE(task, hash, diff):
  outcome, n := task.run()
  win := 0
  PoUW :=  $\perp$ 
  // simulating n Bernoulli tests
  l  $\leftarrow$   $\mathcal{U}[0, 1]$  // query SGX RNG
  if  $l \geq 1 - (1 - \textit{diff})^n$  then
    PoUW =  $\Sigma_{\text{intel}}(\textit{hash} \parallel \textit{diff} \parallel 1)$ 
  fi
  return outcome, PoUW

```

Figure 5.8: PoUW Runtime.

Another challenge is to ensure the result of instruction counting is used properly—we cannot rely on the useful work programs themselves. The solution is to wrap the useful work with a predefined, trusted PoUW runtime, and make sure to the enclave can only be entered through the PoUW runtime. The logic of PoUW runtime is summarized in Fig. 5.8, and it is denoted as `PoUWruntime.so` in Fig. 5.7. The PoUW runtime serves as an “in-enclave” loader that launches the useful work program with proper input and collects the result of instruction counting. It takes the block hash and difficulty and starts mining by running the mining program. Once the mining program returns, the PoUW runtime extracts the instruction counter from the reserved register. Then it draws a random value from SRNG and determines whether a new block should be generated, based on the instruction counter and the current difficulty. If a block should be generated, the PoUW runtime produces an attestation recording the template hash that it is called with and the difficulty.

The last step of the toolchain is to compile the resultant assembly and link it (using linker `GNU ld`) with the PoUW runtime (`PoUWruntime.so`), to produce

```

...
.LEHBE0:
    leaq    1(%r15), %r15    # added by PoUW
    call   _ZN11stlpmx_std12basic_stringIcNS...
.LEHE0:
.loc 7 70 0 is_stmt 0 discriminator 2
    leaq    3(%r15), %r15    # added by PoUW
    leaq   -80(%rbp), %rax #, tmp94
    movq   %rax, %rsi # tmp94,
    movq   %rbx, %rdi # _4,
.LEHB1:
    leaq    1(%r15), %r15    # added by PoUW
    call   _ZN11stlpmx_std12out_of_rangeC1ER...
.LEHE1:
...

```

Figure 5.9: A snippet of assembly code instrumented with the REM toolchain. Register r15 is the reserved instruction counter; it is incremented at the beginning of each basic block in the lines commented added by PoUW.

the PoUW enclave. Figure 5.9 shows a snippet of instrumented assembly code. This PoUW enclave is finally signed by an Intel SGX signing tool, creating an application PoUWEnclave.so that is validated for loading into an enclave.

The security of instruction counting relies on the assumption that once instrumented, the code cannot alter its behavior. To realize this assumption in SGX, we need to require two invariants. First, code pages must be non-writable; second, the useful work program must be single threaded.

Enforcing Non-Writable Code Pages Writable code pages allow a program to rewrite itself at runtime. Although necessary in some cases (e.g. JIT), writable code opens up potential security vulnerabilities. In particular, writable code pages are not acceptable in REM because they would allow a malicious useful work program to easily bypass the instrumentation. A general memory protection policy would be to require code pages to have $W \oplus X$ permission, namely to be either writable or executable, but not both. However, $W \oplus X$ permissions

are not enforced by the hardware. Intel has in fact acknowledged this issue [13] and recommended that enclave code contain no relocation to enable the $W\oplus X$ feature.

REM thus explicitly requires code pages in the enclave code (`usefulwork.so`) to have $W\oplus X$ permission. This is straightforward to verify, as with the current implementation of the SGX loader, code page permissions are taken directly from the ELF program headers [12].

Enforcing Single Threading Another limitation of SGX is that the memory layout is largely predefined and known to the untrusted application. For example, the State Save Area (SSA) frames are a portion of stack memory that stores the execution context when handling interrupts in SGX. This also implies that the SSA pages have to be writable. The address of SSA frames for an enclave is determined at the time of initialization, as the Thread Control Structure (TCS) is loaded by the untrusted application through an EADD instruction. In other words, the address of SSA is always known to the untrusted application. This could lead to attacks on the instruction counting if a malicious program has multiple threads that interact via manipulation of the execution context in SSA. For example, as we will detail later, REM stores the counter in one of the registers. When one thread is swapped out, the register value stored in an SSA is subject to manipulation by another thread.

While more complicated techniques such as Address Space Layout Randomization (ASLR) for SGX could provide a general answer to this problem, for our purposes it suffices to enforce the condition that an enclave can be launched by at most one thread. As an SGX enclave has only one entry point, we can

```

.section data
ENCLAVE_MTX:
.long 0

.section text
...
enclave_entry:
xor %rax, %rax
xchgl ENCLAVE_MTX(%rip), %rax
cmp %rax, 0
jnz enclave_entry

```

Figure 5.10: Code snippet: a spinlock to allow only the first thread to enter `enclave_entry`

instrument the code with a spinlock to allow only the first thread to pass, as shown in Fig. 5.10.

Known entry points REM expects the PoUW toolchain and compliance checker to provide and verify a subset of Software Fault Isolation (SFI), specifically indirect control transfers alignment [105, 176, 256, 149]. This ensures that the program can only execute the instruction stream parsed by the compliance checker, and not jump to the middle of an instruction to create its own alternate execution that falsifies the instruction count. Our implementation does not include SFI, as off the shelf solutions such as Google’s Native Client could be integrated with the PoUW toolchain and runtime with well quantified overheads [256].

5.5.3 Hierarchical Attestation

A blockchain participant that verifies a block has to check whether the useful work program that produced the block’s PoUW followed the protocol and correctly counted its instructions. SGX attestations require such a verifier to obtain a fingerprint of the attesting enclave. As we allow arbitrary work, a naïve im-

plementation would store all programs on the blockchain. Then a verifier that considers a certain block would read the program from the blockchain, verify it correctly counts instructions, calculate its fingerprint, and check the attestation. Beyond the computational effort, just placing all programs on the blockchain for verification would incur prohibitive overhead and enable DoS attacks via spamming the chain with overly large programs. The alternative of having an entity that verifies program compliance is also unacceptable, as it puts absolute blockchain control in the hands of this entity: it can authorize programs that deterministically win every execution.

To resolve this predicament, we form PoUW attestations with what we call *two-layer hierarchical attestations*. We hard-code only a single program's fingerprint into the blockchain, a static-analysis tool called *compliance checker*. The compliance checker runs in a trusted environment and takes a user-supplied program as input. It validates that it conforms with the requirements defined above. First, it confirms the text section is non-writable. Then it validates the work program's compliance by disassembling it and confirming that the dedicated register is reserved for instruction counting and that counts are correct and appear where they should. Next, it verifies that the PoUW runtime is correctly linked and identical to the expected PoUW runtime code. Finally, it verifies the only entry point is the PoUW runtime and that this is protected by a spinlock as shown in Fig. 5.10. Finally, it calculates the program's fingerprint and outputs an attestation including this fingerprint.

Every PoUW then includes two parts: The useful work program attestation on the mining success, and an attestation from the compliance checker of the program's compliance (Figure 5.11). Note that the compliance attestation and the

program's attestation must be signed by the same CPU. Otherwise an attacker that compromises a single CPU could create fake compliance attestations for invalid tasks. Such an attacker could then create blocks at will from different uncompromised CPUs, circumventing the detection policy of Section 5.4.

In summary, the compliance enclave is verified through the hard-coded measurement in the blockchain agent. Its output is a measurement that should be identical to the measurement of the PoUW enclave `PoUWEnclave`. So, PoUW Enclave's output should match the block template (namely the hash of the block prefix, up to the proof) and the prescribed difficulty.

Generalized Hierarchical Attestation The hierarchical attestation approach can be useful for other scenarios where participants need to obtain attestations to code they do not know in advance. As a general approach, one hard-codes the fingerprint of a *root compliance checker* that verifies its children's compliance. Each of them, in turn, checks the compliance of its children, and so on, forming a tree. The leaves of the tree are the programs that produce the actual output to be verified. A hierarchical attestation therefore comprises a leaf attestation and a path to the root compliance checker. Each node attests the compliance of its child.

5.5.4 IAS access overhead

Verifying blocks doesn't require trusted hardware. However, due to a design choice by Intel, miners must contact the IAS to verify attestations. Currently there is no way to verify attestations locally. This requirement, however, does

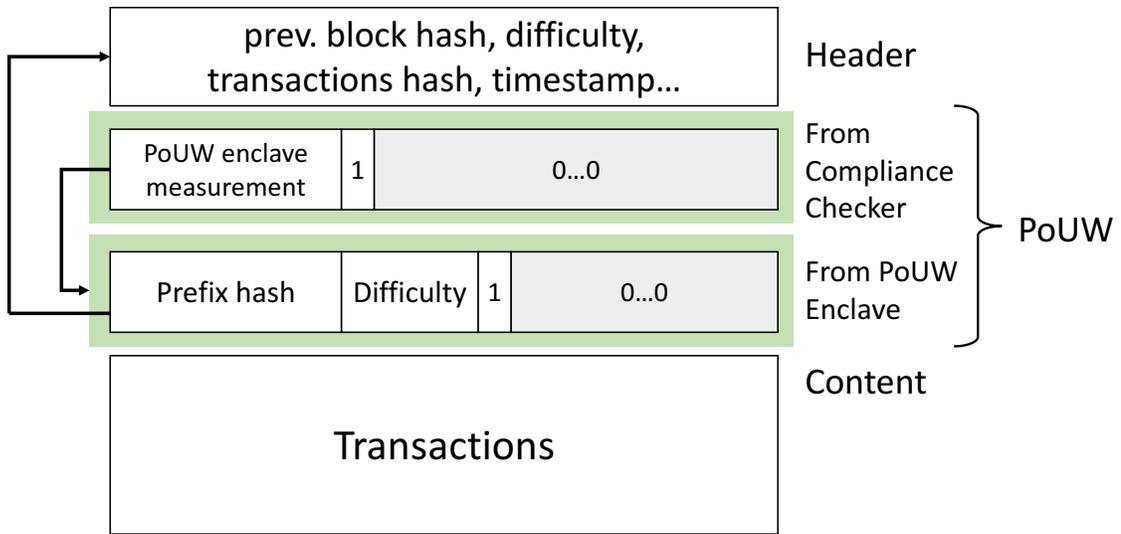


Figure 5.11: Block structure with a proof comprising the quotes from the compliance enclave and a work enclave.

not change the basic security assumptions. Moreover, a simple modification to the IAS protocol, which is being tested by Intel [4], could get rid of the reliance on IAS completely on verifiers' side.

Recall that the IAS is a public web service that receives SGX attestations and responds with verification results. Requests are submitted to the IAS over HTTPS; a response is a signed "report" indicating the validation status of the queried platform [154]. In the current version of IAS, a report is not cryptographically linked with its corresponding request, which makes the report only trustworthy for the client initiating the HTTPS session. Therefore an IAS access is required for every block verification by every blockchain participant.

However, the following modification can eliminate this overhead: simply echoing the request in the body of the report. Since the report is signed by Intel using a published public key [154, 155], only one access to IAS would be needed globally for every new block. Other miners could use the resulting signed report. Such a change is under testing by Intel for future versions of the IAS [4].

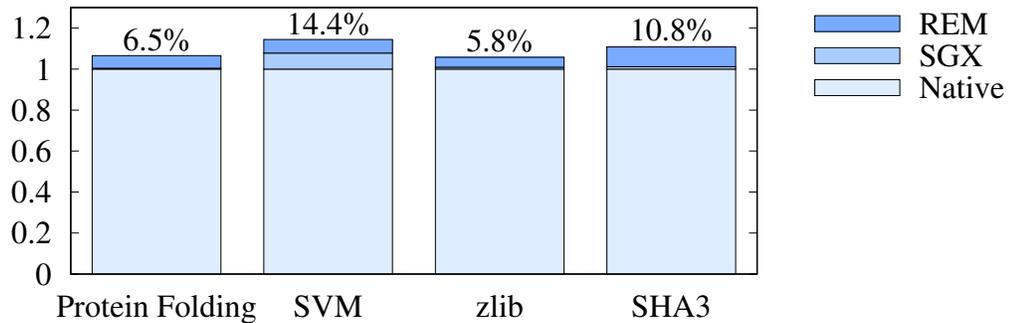


Figure 5.12: REM Overhead.

5.5.5 Experiments

We evaluate the overhead of REM with four examples of useful work benchmarks in REM as mining programs: a protein folding algorithm [134], a Support Vector Machine (SVM) classifier [77], the zlib compression algorithm (iterated) [1], and the SHA3-256 hash algorithm (iterated) [21]. We evaluate each benchmark in three modes:

Native We compile with the standard toolchain.

SGX We port to SGX by removing system calls and replacing system libraries with SGX-compliant ones. Then we compile in SGX-prerelease mode and run with the SGX driver v1.7 [153].

REM After porting to SGX, we instrument the code using our REM toolchain. We then proceed as in the SGX mode.

We use the same optimization level (`-O2`) in all modes. The experiments are done on a Dell Precision Workstation with an Intel 6700K CPU and 32GB of memory.

We compared the running time in three modes and the results are shown

in Fig. 5.12. The running time of the native mode is normalized to one as a baseline. For all four useful workloads, we observe a total overhead of 5.8% ~ 14.4% in REM relative to the native mode. Because the code is instrumented at control flow transfers, workloads with more jumps will incur more counting overhead. For example, SHA3-256 is highly iterative compared with the other workloads, so it incurs the most counting overhead.

We note that overhead for running in SGX is not uniform. For computation-bound workloads such as protein folding, zlib, and SHA3, SGX introduces little overhead ($< 1\%$) because the cost of switching to SGX and obtaining attestations is amortized by the longer in-enclave execution time of the workload. In the shorter SVM benchmark, the cost of entering SGX is more significant.

In summary, we observe an overhead of roughly 5 – 15% for converting useful-work benchmarks into REM PoUW enclave.

5.6 Waste Analysis

To compare PoUW against PoET and alternative schemes, we explore a common game-theoretic model (with details deferred to the appendix). We consider a set of operators / agents that can either work locally on their own useful workloads or utilize their resource for mining. Based on the revenue from useful work and mining, and the capital and operational costs, we compute the equilibrium point of the system. We calculate the waste in this context as the ratio of the total resource cost (in U.S. dollars) spent per unit of useful work on a mining node compared with the cost when mining is not possible and all operators do useful work. We plug in concrete numbers for the parameters based on statistics we

collected from public data sources.

Initial study of PoET identified a subtle pitfall involving miner’s ability to mine simultaneously on multiple blockchains, a problem solved by Milutinovic et al. [187] in a scheme we call *Lazy-PoET*. Our analysis, however, reveals that even Lazy-PoET suffers from what we call the *stale-chip problem*. Miners are better off maintaining farms of cheap, outdated CPUs just for mining than using new CPUs for otherwise useful goals.

We consider instead an approach in which operators utilize their CPUs while mining, making newer CPUs more attractive due to the added revenue from the useful work done. We call this scheme *Busy PoET*. We find that it improves on Lazy Poet, but remains highly wasteful.

This observation leads to another approach, *Proof of Potential Work (PoPW)*. PoPW is similar to Busy-PoET, but reduces mining time according to the speed of the CPU (its potential to do work), and thus rewards use of newer CPUs. Although PoPW would greatly reduce waste, SGX does not allow an enclave to securely retrieve its CPU model, making PoPW theoretical only.

We conclude that PoUW incurs the smallest amount of waste among the options under study. Below we present our model, parameter choices, and analyses of the various mining schemes.

5.6.1 Resource Consumption Model

In this appendix, we present a general economic / game-theoretic model for modeling resource consumption of consensus schemes. This model guides us

toward an understanding of optimal mining strategies for various consensus schemes and thus a basis for comparison among them. We detail the model in Section 5.6.1.1. We estimate real-world parameter choices for this model in Section 5.6.1.2 and in Section 5.6.1.3 present *cost per unit of useful work*, our key metric of waste / resource consumption. We use this model in Appendix 5.6.2 to compare various SGX-based mining schemes show that REM results in less resource waste than alternatives such as PoET.

5.6.1.1 Model

In our model, we consider a set of N^{op} operators that choose to commit their CPUs either to mining or unrelated useful work. This reflects the fact that there are certain barriers to enter the mining industry and therefore not everyone will participate. For completeness we also discussed the implication of removing the limit of numbers of operators in Section 5.6.2.7.

Each operator has an annual budget of budget for purchasing hardware and for paying operating expenses. In our context, both expenses are a function of the CPUs the operator chooses to use. Denote by age the age in years of the CPUs maintained by the operator, which we assume for simplicity to be uniform. If an operator chooses to maintain new CPUs, she enjoys better performance and efficiency (computation per power unit), but the cost is higher. The latest CPU has an age of 0, and we arbitrarily set the oldest CPU available to 10, denoted $\text{age}_{\text{max}} = 10$. Choosing a higher age_{max} value or removing this limit strengthens our results. In some situations, the operator may choose to have the CPUs but not utilize them. Denote by $u \in [0, 1]$ the utilization level of a CPU, where 0 means idle and 1, fully utilized.

The annual capital cost of maintaining a CPU of age age is denoted by the function $C(\text{age})$. The function decreases with CPU age, as older CPUs are significantly cheaper from recycling marketplace. The annual energy cost of a single CPU is denoted by the function $E(u)$, which increases with u . Denote by $\eta(\text{age}) \in [0, 1]$ the performance slowdown of a CPU, normalized to that of the latest model in the same family. η increases with CPU age age , as newer generations (of similar MSRP) tend to be more powerful than their antecedents. Beyond the CPU itself, there is an overhead for the platform on which it runs. Denote by O_{std} the annual cost overhead for running a CPU, including server, racking, cooling etc. In some schemes (e.g., PoW), an operator can reduce costs by placing the CPU in a farm—a dedicated platform stripped down to essential resources and thus usable only for mining. Denote by O_{farm} (naturally $O_{\text{farm}} < O_{\text{std}}$) the annual cost overhead for running a CPU in a dedicated mining farm.

Due to performance improvements, the annual income for useful work from a CPU is a function $R_w(u, \text{age})$ of both its age and utilization (or simply R_w when the parameters are clear from the context). The function increases with both parameters. We assume operators have unbounded useful work, enough to populate any and all hardware they can afford.

We let R_{annual} denote the annual total mining revenue of the system, which we assume is independent of any other variable and vary our analyses. This total revenue is divided among the participants in a manner that depends on the specific scheme used, and the details are given below.

Operators strive to maximize their net revenue by choosing from a space of three strategies that dictate hardware use. These are:

1. *No mining*: The operator chooses not to mine, using her CPUs solely for useful work. This is profitable as long as the income R_w offsets costs.
2. *Standard mining*: The operator uses standard servers and mines on them (if the scheme allows).
3. *Farming*: The operator uses farm machines for mining, reducing the per-CPU overhead, but losing the ability to perform useful work.

We adopt a population-based representation of the strategy choices made by operators. Specifically, for operators collectively implementing a hybrid strategy, we express their choice in terms of the aggregate fraction of resources devoted to each strategy by the full population.

To compare the resource waste associated with different consensus systems, we examine the optimal operation point for each. We model operators' utility, and identify the best strategy for a rational operator. We develop expressions for the operators' revenues, and instantiate them using the values obtained as explained in Section 5.6.1.2. We then calculate the equilibrium point of each scheme and the optimal operation point for miners in each scheme. We measure the waste of each scheme in terms of the cost for each unit of useful work, referred to as the *useful price*.

5.6.1.2 Parameter Values

To estimate the price, performance, and energy efficiency of a CPU as a function of its age, we investigate historical CPU data for a family of 7 Intel CPUs spanning ten years. From each generation of Intel CPUs since 2006, we picked the fastest desktop chip. For each CPU, we estimate its price as of 2016 according to

| Generation | Sample CPU | Launch Date | Q3'16 Price | speed (normed) | Power at $u = 1$ |
|--------------|------------------|-------------|-------------|----------------|------------------|
| Cedarmill | Pentium 4 661 | Q1'06 | \$5 | $\cdot 3$ | 85W |
| Wolfdale | Core 2 Duo E8500 | Q3'08 | \$10 | 0.21 | 65W |
| Lynnfield | Core i7-880 | Q2'10 | \$35 | 0.52 | 95W |
| Sandy Bridge | Core i7-2700K | Q4'11 | \$50 | 0.80 | 95W |
| Ivy Bridge | Core i7-3770K | Q2'12 | \$75 | 0.87 | 77W |
| Haswell | Core i7-4770K | Q2'13 | \$150 | 0.89 | 84W |
| Skylake | Core i7-6700K | Q3'15 | \$290 | 1.00 | 91W |

Table 5.1: Sample CPUs and specifications. Benchmark scores are normalized by the score of i7-6700K.

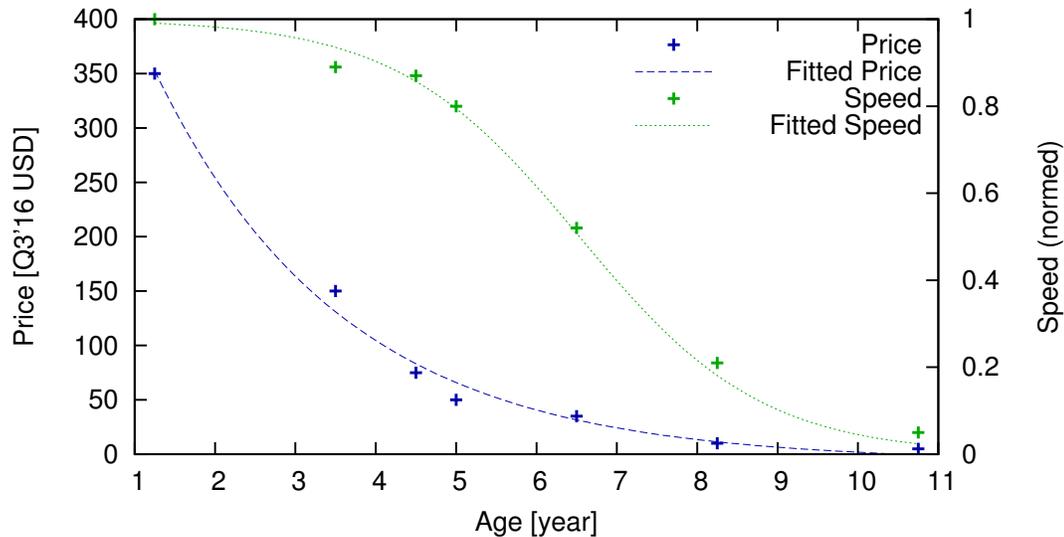


Figure 5.13: CPUs' price and speed as function of age.

several marketplaces worldwide [9] and its performance according to standard benchmarks [16]. We present CPU specifications as published by Intel. Long discontinued models can still be purchased from many suppliers, generally at very low per-unit costs for high-volume orders. All amounts are given in 2016 USD, denoted by \$.

Price and performance Table 5.1 summarizes CPU specs. From the data in Table 5.1 we can see clearly that the CPU price C drops exponentially as it ages.

CPU performance manifests a segmented, exponential trend: It doubled each year between 2006 and 2011 (following Moore’s Law), but then improvements slowed to around 10% per year. Our model uses a logit function to approximate this trend, as is shown in Figure 5.13. As a result of the performance slowdown of older CPUs, R_w decreases with age in a similar way.

Useful work revenue We used the cost model of Amazon Web Services to approximate the useful-work revenue R_w . In particular, we assume a miner’s mining computer has performance similar to an m4.large EC2 instance, which has 2 CPU cores and 8 GiB of memory [3]. As of the time of writing, an m4.large instance costs 0.12 per hour [2].

5.6.1.3 Cost per unit of useful work

To compare the resource costs of the different schemes in Section 5.6.2, we will compute for each its *cost per unit of useful work*.

As a baseline, we first compute the cost per unit of useful work on an optimal hardware setup devoted exclusively to useful work. The revenue of a single CPU for useful work is the income over a year from useful work (R_w) minus the expenses for hardware purchase (C), overhead (O_{std}), and power (E):

$$R_w(u, \text{age}_w) - (C(\text{age}_w) + O_{\text{std}} + E(u, \text{age}_w)). \quad (5.5)$$

The number of CPUs an operator can purchase is

$$N^{\text{cpu}}(\text{age}_w) = \frac{\text{budget}}{C(\text{age}_w) + O_{\text{std}} + E(u, \text{age}_w)}. \quad (5.6)$$

Therefore the total profit from useful work is

$$P_{\text{useful}}(u, \text{age}_w) = N^{\text{cpu}}(\text{age}_w) \times \left(R_w(u, \text{age}_w) - (C(\text{age}_w) + O_{\text{std}} + E(u, \text{age}_w)) \right) \quad (5.7)$$

For simplicity we assume power usage grows linearly with utilization, and so utilization of 1 is optimal to reduce the waste of the independent elements C and O_{std} . Plugging in values from Section 5.6.1.2, we can optimize equation 5.7, resulting in an optimum CPU age of $\hat{\text{age}}_w = 3.48$ years and a corresponding *baseline cost per unit of work of \$391.25*.

For a population of miners, the overall cost per unit of useful work is equal to the total investment in the mining ecosystem over a year, divided by the number of units of useful work. Consequently, it is a function of the fraction of mining CPUs doing useful work. If 100% of CPU time is devoted to useful work, the cost per unit of useful work is \$391.25; 0% corresponds to infinite cost per unit of useful work. Smaller percentages of CPU investment in useful work correspond to higher costs.

5.6.2 Comparative Analysis of Mining Schemes

Using our model from Appendix Section 5.6.1, we now present an analysis showing that PoUW incentivizes minimally wasteful mining behavior and / or achieves secure consensus more effectively than alternatives. An important part of this section is our presentation of a spectrum of five consensus schemes, including PoW, PoUW, and three other SGX-based schemes (two of them newly presented in this work). These schemes illustrate a range of technical approaches

against which we compare PoUW as a means of validating our design choices.

We first present and analyze our four consensus schemes other than PoUW—what we call *strawman schemes*—in Section 5.6.2.1. We then offer detailed revenue analysis of PoUW in Section 5.6.2.6. Figure 5.14 compares the results for the different schemes at the parameter values calculated in Section 5.6.1.2. We consider a conservative total cryptocurrency revenue of \$20m. At higher values the significance of PoUW would only increase. Figure 5.15 compares the schemes at varying total cryptocurrency revenue values.

5.6.2.1 Strawman Schemes

Here we present our spectrum of strawman consensus schemes that serve as points of comparison with and explanations of design choices in PoUW. We start with the popular Proof of Work (PoW), and then present PoET and a proposed minor variant on PoET, as well as two strawman solutions of our own that demonstrate the difficulty of achieving waste-limited SGX-based consensus. We analyze PoUW in Section 5.6.2.6. The reader may choose to skip directly to Section 5.6.2.6 and then flip back here to understand the alternatives and rationale for our design choices.

5.6.2.2 Proof of Work

We start with PoW, as used in Bitcoin, Ethereum, and other common cryptocurrencies. For the purpose of this analysis, we consider only CPU mining. See below for a discussion of dedicated mining hardware.

| Schemes | Choices | $\hat{\text{age}}_S$ | $\hat{\text{age}}_F$ | f_{2nd} | Waste |
|-------------|---------|----------------------|----------------------|-----------|-------|
| Useful work | U | - | - | - | 1.0 |
| PoW | U, F | - | 4.68 | 76% | 4.2 |
| Lazy-PoET | U, F | - | age_{\max} | 76% | 4.2 |
| Busy-PoET | S, F | 4.51 | age_{\max} | 42% | 2.5 |
| PoPW | S, F | 4.39 | 5.61 | 26% | 1.4 |
| PoUW | U, S | 4.39 | - | 100% | 1.1 |

Figure 5.14: Summary of revenue analysis result with parameters according to Section 5.6.1.2. Notation: U is useful work, S is standard mining, F is farming. f_{2nd} is the ratio of participants choosing the second option, which is the more wasteful one. Waste is the cost for one unit of useful work normalized to the baseline (the cost in a system with no mining): \$391.25 ($\hat{\text{age}}_w = 3.48$).

Each operator therefore has two options: either buying CPUs for useful work, or buying CPUs for farmed mining (i.e. farming). For both scenarios, operators can choose freely to use CPUs of any age. Denote by $f^m \in [0, 1]$ the ratio of operators that choose to mine. Denote by $\hat{\text{age}}_w$ and $\hat{\text{age}}_F$ the optimal age for doing useful work and farming, respectively.

Due to symmetry, the optimal age of CPUs at all miners is the same, and so their expended mining power is the same. The mining revenue is therefore distributed uniformly among all miners, and a single miner's income is simply R_{annual} divided by the total number of mining CPUs, leaving a total mining revenue of

$$P_m(u, \text{age}, f^m) = N^{\text{cpu}}(\text{age}) \cdot \left(\frac{R_{\text{annual}}}{f^m \cdot N^{\text{op}} \cdot N^{\text{cpu}}(\text{age}) \cdot u} - C(\text{age}) - O_{\text{farm}} - E(u, \text{age}) \right). \quad (5.8)$$

Again, utilization is optimized at 1. The stable operation point is when the revenue from mining equals that of honest work (5.9), where operators are not

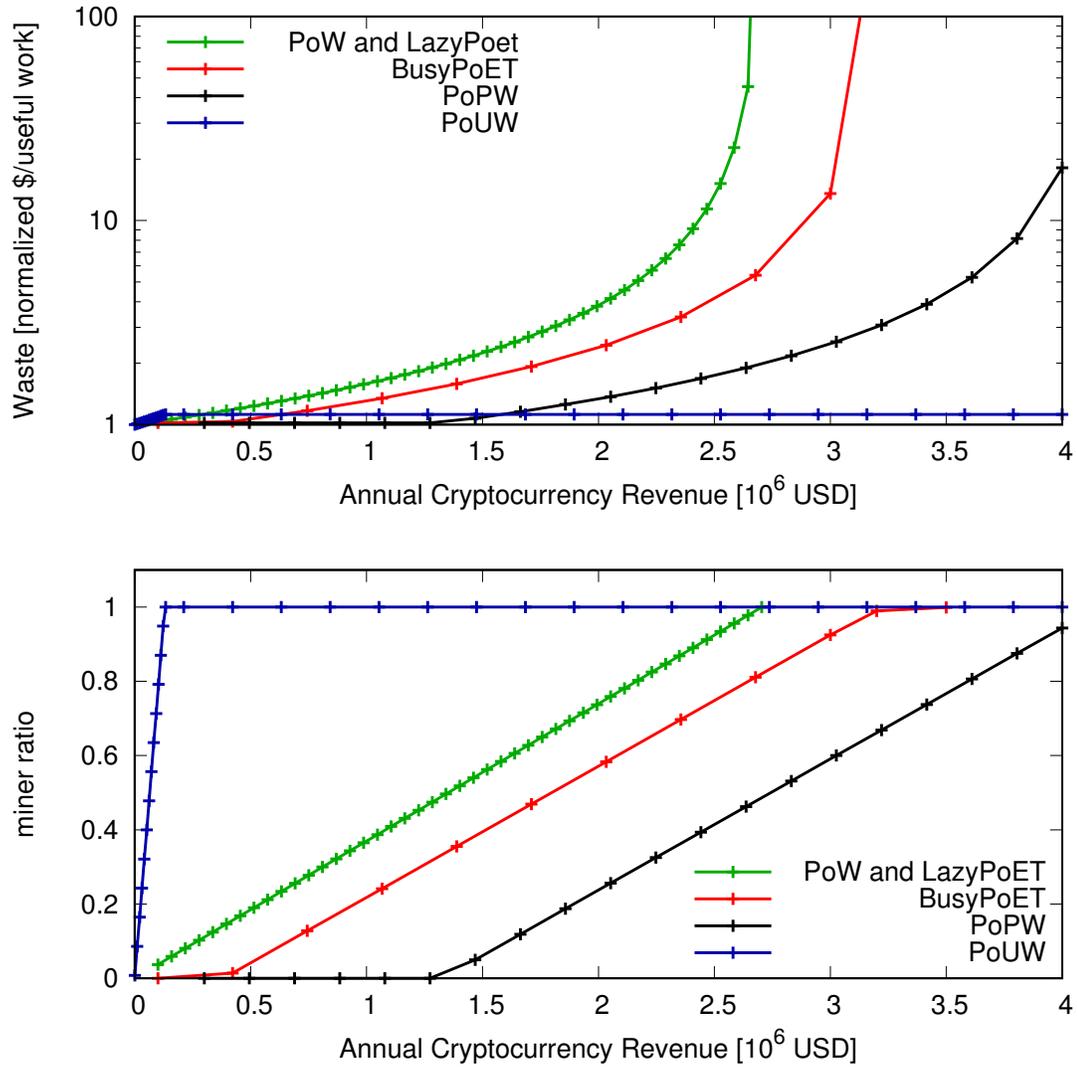


Figure 5.15: Revenue analysis: The x axis is the annual cryptocurrency revenue (R_{annual}) and the left y -axis is the waste factor, i.e. useful price normalized by the useful price without mining. The right y -axis is the ratio of participants choosing the more-wasteful option in their scheme.

motivated to switch sides,

$$P_{\text{useful}}(1, \text{age}_w) = P_m(1, \text{age}_w, f^m) . \quad (5.9)$$

Solving (5.9) we obtain $f^m(\text{age}_m)$ as a function of age_m .

We still need to find the optimal operation parameters for mining. To this

end, we calculate the symmetric Nash equilibrium where if all operators who choose to mine at age age , a single operator cannot increase her revenue by operating at a different age. In fact, finding such an age amounts to finding an age that if all but one miner operate at age , then the other miner's optimal operation age must also be age , that is,

$$age = \max_{age'} P_m(u, age', f^m(age)) . \quad (5.10)$$

With the numbers from Section 5.6.1.2, we find the optimal age for mining is $age_F = 4.68$ years. At the equilibrium, 76.0% of the operators would be mining. The cost for one unit of useful work is \$1643.25. The results are summarized in Figure 5.14.

Figure 5.15 shows how the annual cryptocurrency revenue (R_{annual}) affects the miner ratio at the equilibrium, f^m , and therefore the cost of useful work. As the value of the currency and hence the mining revenue increases, the ratio of operators choosing to mine increases linearly, and the cost of useful work increases exponentially. As the miner ratio reaches 100%, the useful price goes to infinity as nobody is doing useful work.

Dedicated Hardware. Our analysis above assumes use of a CPU for mining, of course. But PoW functions that have been used for cryptocurrencies for extended periods, namely double-SHA256 (for Bitcoin) and Scrypt (for Litecoin), led to the development of dedicated hardware, particularly Application-Specific Integrated Circuits (ASIC). By design, these are far more efficient than any generic hardware for the purpose of the given PoW, but cannot be used for anything else. Arguably, for any PoW function, dedicated hardware can outperform generic hardware.

The equilibrium point in our analysis, however, is not actually affected by

the performance or cost of dedicated hardware. The reason is that no matter what mining devices are used, a fixed amount of mining revenue is evenly distributed among the mining operators at the equilibrium, leaving the mining revenue unchanged. Therefore, advances in dedicated hardware do not affect the conclusions of our analysis.

5.6.2.3 Proof of Elapsed Time (PoET / Lazy-PoET)

As explained above, Intel proposed PoET as a waste-free PoW replacement for a permissionless blockchain. In PoET, each CPU draws a random number r and sleeps for r time. Whichever gets the smallest number wakes up first and becomes leader for the next consensus epoch. Building on top of Intel's trusted hardware SGX, PoET makes use of the trusted random source protected by hardware, prohibiting selfish actors from increasing the frequency of their blocks, but with minimal computation. But the vanilla PoET proposed by Intel cannot be directly employed. The most critical issue is that it costs nothing for a miner to mine on multiple branches of a blockchain. As has been studied in the context of Proof of Stake, being able to work on multiple branches forces a strong assumption, namely that a majority of the miners blindly follow the protocol, even if each of the members of this majority is not individually motivated to do so. Milutinovic et al. [187] proposed to fix this issue by using SGX's monotonic counters. They argue that depleting all 256 SGX counters ensures that the CPU is tied to a single branch.

We call their patched PoET scheme that maintains the CPU idle while mining *Lazy-PoET*. The point of Lazy-PoET is to allow for mining without any energy waste. In the Lazy-PoET scheme, an operator can use her hardware for either

useful work, with fully-utilized CPUs, or for mining, with idle CPUs. Idle CPUs are cheaper to operate in farming mode, and the mining revenue of an old CPU is the same as that of a new and expensive CPU. Hence, the operator choices are reduced to either farming, or useful-work.

The revenue expression for Lazy-PoET is the same as with Proof of Work (5.8), but here the optimal age is age_{\max} , since there is no benefit in using newer CPUs, and old CPUs are cheaper. Therefore, the revenue is only a function of the miner ratio,

$$P_m^{\text{Lazy}}(f^m) \triangleq P_m(0, \text{age}_{\max}, f^m) . \quad (5.11)$$

Equating the revenues of mining (5.11) and useful work (5.7), we obtain again a miner ratio of 76% and the cost per unit of useful work is \$1643.4, i.e. a waste of 4.2. The results are summarized in Figure 5.14. Note that the numbers are identical to those of PoW, as the formulas are the same modulo age_{\max} . The waste here is due to what we call the stale chip problem — farming with extremely old chips, insufficient for any useful work, yields high revenue. This optimum is robust to mining income changes. Mining is always optimal on stale chips. The annual mining revenue determines the ratio of miners; if it is too small, useful work simply becomes preferable to any sort of mining.

5.6.2.4 Busy Proof of Elapsed Time (Busy-PoET)

With Lazy-PoET, mining CPUs are kept idle. However, we note that the depleted SGX counters are allocated per-enclave, and not for the CPU itself: Different enclave code accesses different SGX counters. This is sufficient to ensure that the SGX is tied to a specific blockchain branch, since the mining enclave for a

blockchain is unchangeable, preventing two from running on the same SGX and depleting the same counters. However, the implication is that the SGX and the CPU itself remain available for any other purpose, while mining and providing proofs of elapsed time as necessary. We call a system with proof-of-elapsed time where miners can concurrently use their hardware for useful work *Busy-PoET*.

Since the mining revenue adds no overhead in Busy-PoET, useful work with mining (henceforth *offhand mining*) is preferable to useful work without it. Nonetheless, farming remains a viable option because stale chips are significantly cheaper than recent ones.

As in previous schemes, the annual revenue R_{annual} is split among all mining CPUs. Denote by f^s , the ratio of operators that are working in standard mode. Unlike the previous schemes, here the CPU count includes the $f^s \times N^{\text{OP}}$ operators in standard mode and the $(1 - f^s) \times N^{\text{OP}}$ operators in farming mode. It is therefore a function of the CPU ages of both standard mining, age_S , and farming, age_F . We defer the development of this expression until after the discussion of both operator modes.

Standard mining Distinguished from PoW, the revenue from standard mining in Busy-PoET has an additional item of useful work revenue R_u :

$$R_w(u, \text{age}_S) + R_m - (C(\text{age}_S) + O_{\text{std}} + E(u, \text{age}_S)). \quad (5.12)$$

The total number of CPUs a standard operator can afford, $N_{\text{std}}^{\text{CPU}}$, has the same expression as in useful work (5.6), yielding a total annual revenue of (the

dependency in f^s is through R_m)

$$P_S(u, \text{age}_S, f^s) = N_{\text{std}}^{\text{cpu}} \times (R_w(u, \text{age}_S) + R_m - (C(\text{age}_S) + O_{\text{std}} + E(u, \text{age}_S))). \quad (5.13)$$

Farming The expression for revenue from farmed mining is similar to that of previous schemes (the dependency in f^s is through R_m),

$$P_F(u, \text{age}_F, f^s) = N_{\text{farm}}^{\text{cpu}} \times (R_m - (C(\text{age}_F) + O_{\text{farm}} + E(u, \text{age}_F))). \quad (5.14)$$

The equilibrium analysis for Busy-PoET is not as simple now, as (5.13) and (5.14) are interdependent through f^s . An equilibrium is a pair $(\text{age}_S, \text{age}_F)$ where an operator cannot improve her revenue by changing her strategy. This amounts to two conditions.

First, she cannot improve her revenue by changing her CPU age (Equation 5.10) where all other operators maintain their strategy. This is expressed for the two operation modes as the equation system

$$\begin{cases} \text{age}_S = \max_{\text{age}'_S} P_S(1, \text{age}'_S, f^s(\text{age}_S, \text{age}_F)) \\ \text{age}_F = \max_{\text{age}'_F} P_F(0, \text{age}'_F, f^s(\text{age}_S, \text{age}_F)). \end{cases} \quad (5.15)$$

Second, she cannot improve her revenue by changing her operation mode. We take this condition,

$$P_{\text{std}}(u, \text{age}_S, \text{age}_F) = P_{\text{farming}}(u, \text{age}_S, \text{age}_F) ,$$

to obtain an expression of f^s as a function of age_S and age_F . The resulting expression is too complex to be placed here.

We numerically solve the equation system and observe that, as with previous schemes, the cost of useful work grows quickly with the annual cryptocurrency revenue. Figure 5.15 shows that the ratio of standard miners decreases, as miners prefer farming with many cheap CPUs over performing useful work – another instance of the stale chip problem.

5.6.2.5 Proof of Potential Work

In order to mitigate the stale chip problem of the PoET variants, we propose a direct solution: Grant more block rewards to CPUs with better performance. Since with this approach a miner proves her CPU power, though she might not be utilizing it, we call it *Proof of Potential Work* (PoPW).

Technically, an SGX program can determine the CPU model by calling the `cpuid` instruction. Based on the CPU model, one can determine the value of a CPU by looking up in a public table, hard-coded in the blockchain protocol. The time for the mining enclave to return is therefore chosen with an exponential distribution parameter that is proportional to the CPU's power. The mining revenue is therefore linear in the slow-down factor of the CPU.

Note that PoPW makes stronger security assumptions than any of the other schemes we discuss. The principal that determines the values of the ever-changing CPU value table has significant power over the blockchain; for example, he can attribute high values to CPUs to which it has better access than other operators.

Standard mining As in Busy-PoET, standard operation dominates useful work, as the revenue for mining comes without any overhead. Recalling that the slowdown of a CPU is $\eta(\text{age})$, the expression for the CPU revenue in standard mode is $R_w(u, \text{age}_S) + R_m \times \eta(\text{age}_S) - (C(\text{age}_S) + O_{\text{std}} + E(u, \text{age}_S))$.

The total number of CPUs an operator can afford has the same expression as in PoW, yielding a total annual revenue of

$$N_{\text{std}}^{\text{cpu}} \times (R_w(u, \text{age}) + R_m \times \eta(\text{age}) - (C(\text{age}_S) + O_{\text{std}} + E(u, \text{age}_S))). \quad (5.16)$$

Farming The CPU revenue for farming is

$$R_m \times \eta(\text{age}_F) - (C(\text{age}_F) + O_{\text{std}} + E(u, \text{age}_F)). \quad (5.17)$$

The expression for number of CPUs is the same as before.

As before, the annual mining revenue is distributed among all mining CPUs, though now proportionally to their slow-down. The equilibrium analysis is similarly to that of Busy-PoET, resulting in a quick increase of waste as the annual cryptocurrency revenue grows, as show in Figure 5.15.

As shown in Figure 5.14, we observe that indeed the stale-chip problem is resolved: now the optimal farming age is 5.61, as opposed to age_{max} in Lazy-PoET and Busy-PoET, but the farming problem remains. Starting at some point, when the annual revenue is high enough, it becomes more profitable to farm, keeping idle CPUs, then to spend that amount on power for useful work. Also as discussed before, the trust model of PoPW is arguably too strong for a decentralized system.

5.6.2.6 Proof of Useful Work

Our solution, *Proof of Useful Work (PoUW)*, avoids the issues with previous schemes by directly counting work done towards mining effort. Since mining revenue is only granted when work is done, farming without useful work means the CPUs must be processing useless work in this mode. Therefore, standard operation dominates farming, as the revenue from useful work comes at no additional cost.

The useful work analysis and optimal age calculation remain unchanged, resulting in a revenue as expressed in (5.7).

For standard operation, the mining revenue now depends on the utilization of the CPU as well as on its age. The useful work utilization suffers a decrease due to the overhead of online effort monitoring, denoted O_{counting} . The mining revenue does not suffer such a reduction, as the overhead is taken into account when calculating the mining effort.

Denote the optimal age for standard operation by age_S . The expression for the number of CPUs is as usual (5.6), at $\text{age} = \text{age}_S$. The standard mining revenue is the product of the CPU count and (5.18). This expression is a function of R_m and age_S , i.e.,

$$R_w\left(\frac{u}{O_{\text{counting}}}, \text{age}_S\right) + R_m \times \eta(\text{age}_S) \times u - (C(\text{age}_S) + O_{\text{std}} + E(u, \text{age}_S)). \quad (5.18)$$

As with PoW, the value of R_m is chosen such that the total revenue distributed is R_{annual} ,

$$R_m = \frac{R_{\text{annual}}}{f^s \times N^{\text{op}} \times N_{\text{std}}^{\text{CPU}} \times u \times \eta}. \quad (5.19)$$

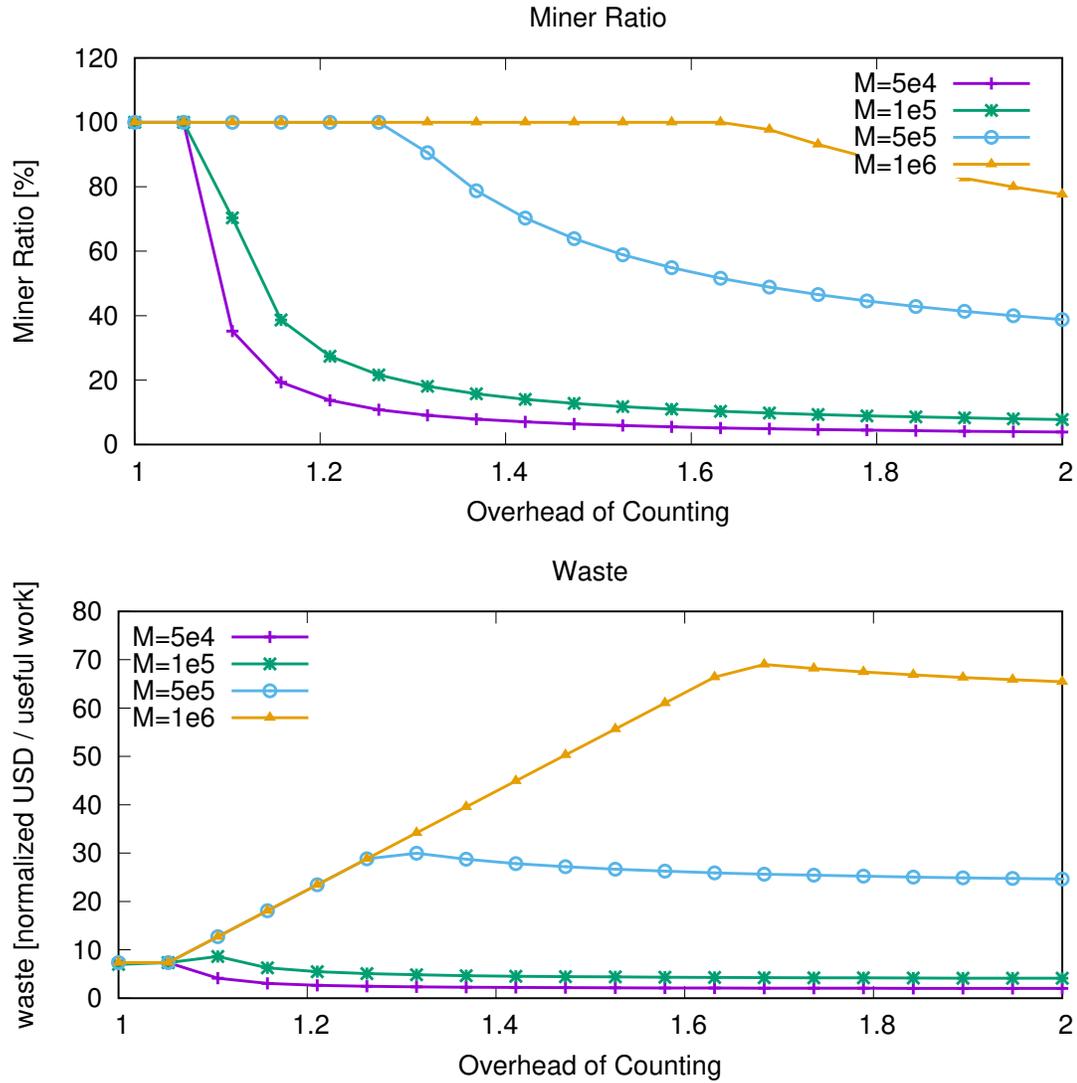


Figure 5.16: O_{counting} affects the miner ratio and the waste in PoUW.

The equilibrium point is where the standard and useful-work revenues are the same. Solving the equation we obtain an expression for the ratio of standard miners at equilibrium, which is a function of age_S . We proceed to find a symmetric Nash equilibrium as is done for PoW.

With the parameter values of Section 5.6.1.2, we find that all operators choose to work in standard mode. The optimal CPU age is 4.39 years, rather than the 3.48 optimal age for useful-work operation. The average cost for useful

work is \$430.1, i.e. a waste of 1.1. The results are summarized in Figure 5.14. As shown in Figure 5.15, PoUW has the lowest waste among the four schemes we compared. Unlike other schemes, the waste in PoUW is not affected by the annual cryptocurrency revenue. The reason for that is PoUW effectively encourages all of the participants to mine as long as the mining revenue isn't critically low. With any reasonable annual cryptocurrency revenue, all of PoUW participants end up mining, yielding a fixed waste per useful work unit.

However, PoUW does introduce the overhead of secure instrumentation (O_{counting}) that is not present in other schemes. The impact of O_{counting} with different annual cryptocurrency revenue is shown in Figure 5.16. The top graph shows how O_{counting} impacts the miner ratio at equilibrium. Given an annual cryptocurrency revenue, higher O_{counting} discourages participants from mining as doing useful work would be more profitable. So the ratio of miners at equilibrium decreases with O_{counting} . The bottom graph shows how O_{std} impacts the waste. Because waste is only incurred by mining, higher annual cryptocurrency revenue increases miner ratio, leading to more waste. Meanwhile, given an annual cryptocurrency revenue, increasing O_{counting} will first increase the waste until all miners are forced to do useful work, where no waste is present.

The conclusion is high O_{counting} could diminish the security of PoUW because of a loss of miner power and an extra waste. As our implementation suggests, REM only incurs a O_{counting} of about 5 – 15%, i.e. 1.05 – 1.15 in Figure 5.16, allowing for high miner participation and low waste.

5.6.2.7 Unbounded analysis

In previous analysis, we assumed a bounded number of operators in the system. This assumption is grounded in the fact that there are certain barriers to enter the mining business, For completeness, we now propose an alternative model where the number of operators is unbounded. In this model there will be infinitely many CPUs doing useful work, and any CPU can switch to that at any point. Weakening this assumption only strengthens the results.

Admittedly, it is tricky to propose a perfect model for such a dynamic and pluralistic system. So we provided a first-order approximation based on realistic but simplifying assumptions. For example, the low electricity cost in China is quite important to the dynamics of Bitcoin mining. but the details of price distribution are unknown. Therefore we assumed the cost of CPUs and electricity is the same for all of the operators.

Just as before, participant has two options besides working on useful work: mining on herself or joining a farm. The age of CPU still plays an important rule as the price and performance vary significantly with the age. A equilibrium is defined similarly as before, satisfying two conditions: 1) revenue for participants is the same as if they were useful workers; and 2) no single operator can earn more by working at a different CPU age.

The analysis for PoW and Lazy-PoET remains basically unchanged because in both schemes operators have in fact only one option besides useful work. We refer readers to the Appendix for more details.

Busy-PoET We note that the incremental overhead of Busy-PoET over useful work and the cost of farming are key to the analysis. Taking it to an extreme

where PoET incurs zero additional overhead, every CPU doing useful work will mine along with the useful work, earning some mining revenue at no incremental cost. On the other hand, if the cost incurred by PoET is high, fewer participants will perform PoET at equilibrium, hence leaving some profit margin for farming. We define the inefficiency of PoET as the incremental overhead over doing useful work normalized by useful work revenue. The efficiency of PoET is one minus that.

Fig. 5.17 shows the number of standard mining CPUs at equilibrium, with PoET efficiency ranging from 60% to 100%, assuming no farming. We argue that in the case where PoET is very efficient, unbounded analysis is actually more realistic, as the equilibrium point derived from the unbounded model requires an impractically large amount of CPUs.

Approximating the situations in Bitcoin, if we assume there are farmers (or attackers) with access to cheap stale chips and electricity, the presence of them could skew the equilibrium significantly. At each point in Fig. 5.17, standard miners operate at the margin where adding one single CPU renders the standard mining a worse option than not participating. Therefore, a single farmer with cheap resources would expel a large amount of standard mining CPUs from the mining pool. We argue that this is a major drawback because it allows attacks at a relatively low cost.

Another factor that would change the equilibrium is the farming cost. In the extreme case where PoET incurs zero overhead, farming is no longer a viable option, unless it's free, because most of the mining revenue would have been harvested by standard miners, leaving farmers too little to cover the farming cost. But on the other hand, if PoET incurs non-negligible overhead, farming becomes

possible if the farming cost is low enough. Fig. 5.18 shows the thresholds for farming cost as functions of PoET efficiency. For any given PoET efficiency, if farming cost is below than the lower threshold, all operators will end up farming at equilibrium; above the higher threshold, all operators will end up performing standard mining. If the farming cost is between the two, then equilibrium will involve a mixture of both.

PoUW As discussed previously, PoUW renders farming irrelevant, eliminating a major source of waste. However, PoUW does introduce an overhead O_{counting} from instructions counting. Fig. 5.19 shows the number of CPUs in PoUW at equilibrium under different PoUW efficiency. Note that the equilibrium won't be skewed by farmers as farming in PoUW is strictly inferior, irrespective of farming cost.

The conclusion here is that if PoET and PoUW are sufficiently efficient, PoET can get rid of the farming issue unless farmers can operate at a very low cost. In this case, however, the bounded model is more realistic because equilibrium in the unbounded model requires an impractically large amount of participating CPUs. On the other hand, if PoET incurs non-negligible overhead, unbounded analysis draws a similar conclusion as the bounded one does: farming remains a viable option and because of that, an attacker can accrue old CPUs to attack at a lower cost.

5.7 Related Work

Cryptocurrencies and Consensus. Modern decentralized cryptocurrencies have stimulated strong interest in Proof-of-Work (PoW) systems [42, 109, 157] as well

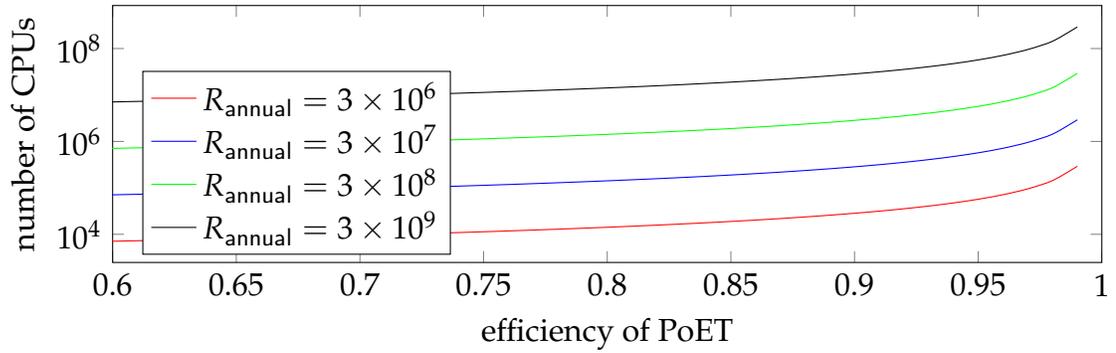


Figure 5.17: The number of standard mining CPUs at equilibrium, as a function of PoET efficiency. R_{annual} denotes the total annual cryptocurrency revenue. As a reference, Bitcoin yielded a total annual revenue of approximately 330 millions in 2015. As PoET efficiency goes to 1, the number of mining CPUs tends to infinity.

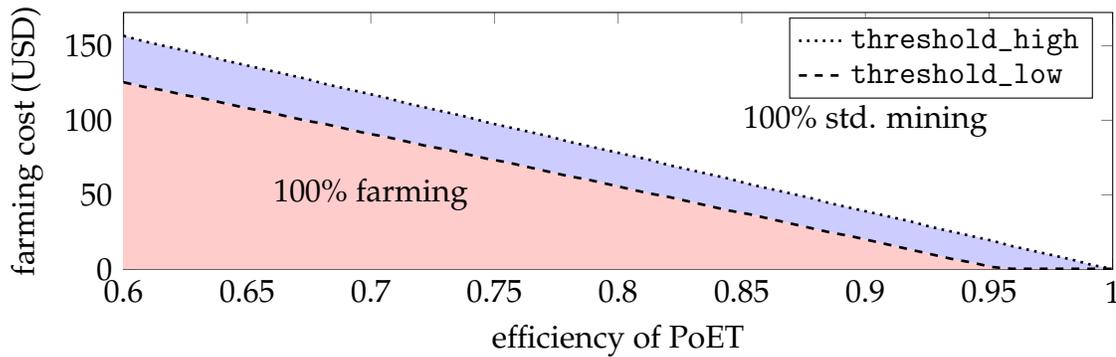


Figure 5.18: Farming cost affects the number of farmers at equilibrium. If the farming cost is below threshold low, all operators will join farms at equilibrium; above threshold high, all operators will do standard mining. Useful work cost is \$391.25.

as techniques to reduce their associated waste.⁴

An approach similar to PoET [87], possibly originating with Dryja [107], is to limit power waste by so-called Proof-of-Idle. Miners buy mining equipment and get paid for proving that their equipment remains idle. Beyond the technical challenges, as in PoET, an operator with a set budget could redirect savings from power to purchase more idle machines, producing capital waste.

⁴“Permissioned” systems, as supported in, e.g., Hyperledger [66] and Stellar [184], avoid waste by using traditional consensus protocols at the cost of avoiding decentralization.

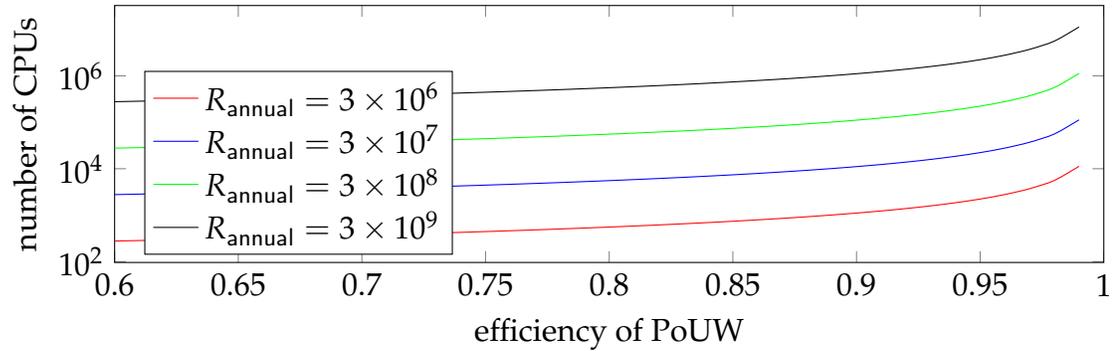


Figure 5.19: The number of standard mining CPUs at equilibrium, as a function of PoUW efficiency (i.e. $1 - O_{\text{counting}}$). R_{annual} denotes the total annual cryptocurrency revenue. As a reference, Bitcoin yielded a total annual revenue of approximately 330 millions in 2015. As PoUW efficiency goes to 1, the number of mining CPUs tends to infinity.

Alternative approaches, like PoUW, aim at PoW producing work useful for a secondary goal. Permacoin [186] repurposes mining resources as a distributed storage network, but recycles only a small fraction of mining resources. Primecoin [167] is an active cryptocurrency whose “useful outputs” are Cunningham and Bi-twin chains of prime numbers, which have no known utility. Gridcoin [137, 136], an active cryptocurrency whose miners work for the BOINC [33] grid-computing network, relies on a central entity. FoldingCoin [218] rewards participants for work on a protein folding problem, but as a layer atop, not integrated with, Bitcoin.

Proof-of-Stake [244, 50, 166, 92] is a distinct approach in which miners gain the right to generate blocks by committing cryptocurrency funds. It is used in experimental systems such as Peercoin [168] and NXT [84]. Unlike PoW, however, in PoS, an attacker that gains majority control of mining resources for a bounded time can control the system forever. PoS protocols also require that funds, used as stake, remain frozen (and unusable) for some time. To remove this assumption, Bentov et al. [51] and Duong et al. [108] propose hybrid PoW / PoS systems.

These works, and the line of hybrid blockchain systems starting with Bitcoin-NG [116, 170, 204], can all utilize PoUW as a low-waste alternative to PoW.

Another line of work on PoW for cryptocurrencies aims at PoWs that resist mining on dedicated hardware and prevent concentration of mining power, e.g., via memory-intensive hashing as in Scrypt [178] and Ethereum [64]. Although democratization of mining power is not our focus here, PoUW in fact achieves this goal by restricting mining to general-use CPUs.

SGX. Due to the complexity of the x86-64 architecture, several works [88, 242, 252] have exposed security problems in SGX, such as side-channel attacks [252]. Tramer et al. [242] consider the utility of SGX if its confidentiality guarantees are broken. Similar practical concerns motivate REM's tolerance mechanism of compromised SGX chips.

Ryoan [149] is a framework that allows a server to run code on private client data and return the output to the client. The (trusted) Ryoan service instruments the server operator's code to prevent leakage of client data. In contrast, in REM, the useful-workload code is instrumented in an *untrusted* environment, and an attestation of its validity is produced within a trusted environment.

Haven [46] runs non-SGX applications by incorporating a library OS into the enclave. REM, in contrast, takes code amenable to SGX compilation and enforces correct instrumentation. In principle, Haven could allow for non-SGX code to be adapted for PoUW.

Zhang et al. [257] and Juels et al. [160] are the first works we are aware of to pair SGX with cryptocurrencies. Their aim is to augment the functionality of smart contracts, however, and is unrelated to the underlying blockchain layer in

which REM operates.

5.8 Conclusion

In this chapter, we presented REM, which supports permissionless blockchain consensus based on a novel mechanism called Proof of Useful Work (PoUW). PoUW leverages Intel SGX to significantly reduce the waste associated with Proof of Work (PoW), and builds on and remedies shortcomings in Intel’s innovative PoET scheme. PoUW and REM are thus a promising basis for partially-decentralized blockchains, reducing waste given certain trust assumptions in a hardware vendor such as Intel.

Using a rigorous analytic framework, we have shown how REM can achieve resilience against compromised nodes with minimal waste (rejected honest blocks). This framework extends to PoET and potentially other SGX-based mining approaches.

Our implementation of REM introduces powerful new techniques for SGX applications, namely instruction-counting instrumentation and hierarchical attestation, of potential interest beyond REM itself. They allow REM to accommodate essentially any desired workloads, permitting flexible adaptation in a variety of settings.

Our framework for economic analysis offers a general means for assessing the true utility of mining schemes, including PoW and partially-decentralized alternatives. Beyond illustrating the benefits of PoUW and REM, it allowed us to expose risks of approaches such as PoET in the use of stale chips, and propose

improved variants, including Proof of Potential Work (PoPW). We found that small changes to the TEE framework would be significant for reduced-waste blockchain mining. In particular, allowing for secure instruction (or cycle) counting would reduce PoUW overhead, and a secure chip-model reading instruction would allow for PoPW implementation.

We reported on a complete implementation of REM, swapped in for the consensus layer in Bitcoin core in a prototype system. Our experiments showed minimal performance impact (5-15%) on example benchmarks. In summary, our results show that REM is practically deployable and promising path to fair and environmentally friendly blockchains in partially-decentralized blockchains.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

To summarize, this thesis explored principled composition of on-chain and off-chain computation to enable provably secure, efficient, and decentralized systems. Chapter 2 and Chapter 3 have presented Town Crier and DECO, respectively, two data oracle protocols that securely connect blockchains to off-chain TLS data sources. Chapter 4 has presented CHURP, the first practical churn-robust secret sharing protocol, which, along with other applications, can endow smart contracts with privacy by storing secrets and computing on privacy data in off-chain committees. Chapter 5 has presented REM and its Proof of Useful Work (PoUW) based consensus protocol, which repurposes off-chain useful computation to achieve on-chain consensus. These systems in this thesis have filled in basic yet missing blockchain needs, addressed key obstacles to their wider adoption, and led to industry adoption.

Looking forward, given the current backlash against overly trusted central hubs, there is an increasing impetus to deploy practical decentralized solutions. However, as blockchain adoption widens, more security and performance challenges will emerge. We are in a unique position to revisit existing centralized systems and meaningfully improve their security and privacy.

Below I outline future research directions.

Towards a holistic understanding of decentralized finance. Blockchains offer appealing features that fit squarely with the needs of financial applications. Various decentralized finance (DeFi) systems, such as decentralized exchanges (e.g., [24, 248]), stablecoins (see [189] for a taxonomy), etc., have been proposed

and deployed, aiming to achieve greater transparency, auditability, and versatility. These systems manage hundreds of millions of dollars in value, but their complexity renders a thorough security analysis challenging.

DeFi faces several security challenges. First, it crucially relies on external data inputs to function. Oracle protocols, including those introduced in Chapters 2 and 3 of this thesis can secure the data channel, but DeFi would make the data *sources* lucrative targets to attack. Off-chain centralized data sources may be corrupted given enough economic incentive, while on-chain ones, e.g., assets prices generated at decentralized exchanges such as [248, 24], are susceptible to various market manipulations. Understanding the security impact of potentially unreliable data sources and designing robust data oracles for DeFi remain a challenge.

Moreover, fairness, i.e., informally the assurance that all users will receive a guaranteed, equal quality of service to the extent that's possible or permitted by the terms of use, is often an overlooked but crucial requirement for decentralized systems, especially for DeFi. For instance, failures to guarantee fair ordering of users' transactions in decentralized exchanges have caused users to lose funds to arbitrage (e.g., see [93] for a study on the widespread and rising deployment of arbitrage bots in blockchain systems). In a recent work [164], we looked into the fair ordering challenge and its connection with traditional consensus algorithms. Designing and building fair consensus protocols and decentralized financial systems are exciting future works. Last but not least, the lack of privacy on blockchains severely limits what DeFi can do. My work on Ekiden [79] achieves this using TEEs. Designing efficient cryptographic protocols to support confidential DeFi remains largely unexplored. I've made some initial progress

with Chapter 3.

Principled trusted setup for trusted execution environments (TEEs). The area of trusted computing has seen rapid development in recent years as major vendors and open-source communities increase their investment in a diverse set of TEE implementations. TEEs are appealing in practice as they achieve strong security with a minimal performance overhead, whereas cryptographic protocols are usually orders of magnitude slower. But TEEs are not a panacea. There are several important issues to be addressed.

Most critically, the security of current TEEs builds on a shaky foundation. For example, attestation is the process for TEE users to cryptographically verify the integrity and identity of an remote TEE. Right now, chip manufacturers are the root of trust, as they're responsible generating, managing, and provisioning attestation keys. As one of the most crucial and security-sensitive components of the entire TEE ecosystem, existing attestation protocols, however, is mostly opaque and varies from vendor to vendor. The lack of a standard, well-scrutinized, auditable remote attestation protocol will be the Achilles heel of the entire ecosystem. Moreover, as open-source TEEs (such as Keystone [174]) are developed and adopted, there is an opportunity to usher in practical changes.

Rethinking digital identity. Identity management lies at the heart of any user-facing system, be it a social media platform, online game, or collaborative tool. Amid backlash against the abuse of personal information by large tech firms [85], new approaches to identity management are emerging. One increasingly popular notion is *decentralized* or *self-sovereign*, where users gather and manage their own credentials under the banner of self-created decentralized identifiers (DIDs). By

controlling private keys associated with DIDs, users are empowered to disclose or withhold their credentials as desired in online interactions. Proposed DID systems (e.g., [86]), however, largely fail to address important technical and usability challenges, such as key management, privacy, legacy compatibility, and uniqueness. In an ongoing project, I'm exploring principled design of DID systems that overcome these problems, building on my previous works. For instance, self-sovereign identity systems as designed would face bootstrap challenge as they require the use of special-purpose signed credentials that are not widely available, if at all. Oracle protocols introduced in this thesis can enable DID systems to leverage the rich data on users available in existing web services. Moreover, the dynamic-committee secret sharing protocol in this thesis can be used to build user-friendly private key management schemes that are suitable to use in decentralized environments.

APPENDIX A
TOWN CRIER

A.1 Formal Modeling

A.1.1 SGX Formal Modeling

As mentioned earlier, we adopt the UC model of SGX proposed by Shi et al. [230]. In particular, their abstraction captures a subset of the features of Intel SGX. The main idea behind the UC modeling by Shi et al. [230] is to think of SGX as a trusted third party defined by a global functionality \mathcal{F}_{sgx} (see Figure 2.3 of Section 2.4.3).

Modeling choices. For simplicity, the \mathcal{F}_{sgx} model currently does not capture the issue of revocation. In this case, as Shi et al. point out, we can model SGX’s group signature simply as a regular signature scheme Σ_{sgx} , whose public and secret keys are called “manufacturer keys” and denoted pk_{sgx} and sk_{sgx} (i.e., think of always signing with the 0-th key of the group signature scheme). We adopt this notational choice from [230] for simplicity. Later when we need to take revocation into account, it is always possible to replace this signature scheme with a group signature scheme in the modeling.

The $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})$ functionality described by Shi et al. [230] is a global functionality shared by all protocols, parametrized by a signature scheme Σ_{sgx} . This global \mathcal{F}_{sgx} is meant to capture all SGX machines available in the world, and keeps track of multiple execution contexts

for multiple enclave programs, happening on different SGX machines in the world. For convenience, this paper adopts a new notation $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})[\text{prog}_{\text{encl}}, \mathcal{R}]$ to denote one specific execution context of the global \mathcal{F}_{sgx} functionality where the enclave program in question is $\text{prog}_{\text{encl}}$, and the specific SGX instance is attached to a physical machine \mathcal{R} . (As the *Relay* in TC describes all functionality outside the enclave, we use \mathcal{R} for convenience also to denote the physical host.) This specific context $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})[\text{prog}_{\text{encl}}, \mathcal{R}]$ ignores all parties' inputs except those coming from \mathcal{R} . We often omit writing (Σ_{sgx}) without risk of ambiguity.

Operations. \mathcal{F}_{sgx} captures the following features:

- *Initialize.* Initialization is run only once. Upon receiving `init`, \mathcal{F}_{sgx} runs the initialization part of the enclave program denoted `outp := progencl.Initialize()`. Then, \mathcal{F}_{sgx} attests to the code of the enclave program `progencl` as well as `outp`. The resulting attestation is denoted σ_{att} .
- *Resume.* When `resume` is received, \mathcal{F}_{sgx} calls `progencl.Resume` on the input parameters denoted `params`. \mathcal{F}_{sgx} outputs whatever `progencl.Resume` outputs. \mathcal{F}_{sgx} is stateful, i.e., allowed to carry state between `init` and multiple `resume` invocations.

Finally, we remark that this formal model by Shi et al. is speculative, since we know of no formal proof that Intel's SGX does securely realize this abstraction (or realize any useful formal abstraction at all for that matter)—in fact, available public documentation of SGX does not provide sufficient information for making such formal proofs. As such, the formal model in [230] appears to be the best available tool for us to formally reason about security for SGX-based

protocols. Shi et al. leave it as an open question to design secure processors with clear formal specifications, such that they can be used in the design of larger protocols/systems supporting formal reasoning of security. We refer the readers to [230] for a more detailed description of the UC modeling of Intel SGX.

A.1.2 Blockchain Formal Modeling

Our protocol notation adopts the formal blockchain framework recently proposed by Kosba et al. [171]. In addition to UC modeling of blockchain-based protocols, Kosba et al. [171] also design a modular notational system that is intuitive and factors out tedious but common features inside functionality and protocol wrappers (e.g., modeling of time, pseudonyms, adversarial reordering of messages, a global ledger). The advantages of adopting Kosba et al.’s notational system are these: the blockchain contracts and user-side protocols are intuitive on their own and they are endowed with precise, formal meaning when we apply the blockchain wrappers.

Technical subtleties. While Kosba et al.’s formal blockchain model is applicable for the most part, we point out a subtle mismatch between their formal blockchain model in [171] and the real-world instantiation of blockchains such as Ethereum (and Bitcoin for that matter). The design of Town Crier is secure in a slightly modified version of the blockchain model that more accurately reflects the real-world Ethereum instantiation of a blockchain.

As we will see in the proof of Theorem 2.2, we must carefully handle the case of **Deliver** arriving after **Cancel**. In the formal blockchain model proposed

by Kosba et al. [171], we can easily get away with this issue by introducing a timeout parameter T_{timeout} that the requester attaches to each datagram request. If the datagram fails to arrive before T_{timeout} , the requester can call **Cancel** any time after $T_{\text{timeout}} + \Delta T$. On the surface, this seems to ensure that no **Deliver** will be invoked after **Cancel** assuming Town Crier is honest.

However, we do not adopt this approach due to a technical subtlety that arises in this context—again, the fact that the Ethereum blockchain does not perfectly match the formal blockchain model specified by Kosba et al [171]. Specifically, the blockchain model by Kosba et al. assumes that every message (i.e. transaction) will be delivered to the blockchain by the end of each epoch and that the adversary cannot drop any message. In practice, however, Ethereum adopts a dictatorship strategy in the mining protocol, and the winning miner for an epoch can censor transactions for this specific epoch, and thus effectively this transaction will be deferred to later epochs. Further, in case there are more incoming transactions than the block size capacity of Ethereum, a backlog of transactions will build up, and similarly in this case there is also guaranteed ordering of backlogged transactions. Due to these considerations, we defensively design our Town Crier contract such that $\$G_{\text{max}}$ -gas sustainability is attained for Town Crier even if the **Deliver** transaction arrives after **Cancel**.

A.2 Proofs of Security

This section contains the proofs of the theorems we stated in Section 2.8.

Theorem 2.1 (Authenticity). *Assume that Σ_{sgx} and Σ are secure signature schemes. Then, the full TC protocol achieves authenticity of data feed under Definition 2.2.*

Proof. We show that if the adversary \mathcal{A} succeeds in a forgery with non-negligible probability, we can construct an adversary \mathcal{B} that can either break Σ_{sgx} or Σ with non-negligible probability. We consider two cases. The reduction \mathcal{B} will flip a random coin to guess which case it is, and if the guess is wrong, simply abort.

- Case 1: \mathcal{A} outputs a signature σ that uses the same pk_{TC} as the SGX functionality \mathcal{F}_{sgx} . In this case, \mathcal{B} will try to break Σ . \mathcal{B} interacts with a signature challenger Ch who generates some $(\text{pk}^*, \text{sk}^*)$ pair, and gives to \mathcal{B} the public key pk^* . \mathcal{B} simulates \mathcal{F}_{sgx} by implicitly letting $\text{pk}_{\text{TC}} := \text{pk}^*$. Whenever \mathcal{F}_{sgx} needs to sign a query, \mathcal{B} passes the signing query onto the signature challenger Ch.

Since $\text{data} \neq \text{prog}_{\text{encl}}(\text{params})$, \mathcal{B} cannot have queried Ch on a tuple of the form $(_, \text{params}, \text{data})$. Therefore, \mathcal{B} simply outputs what \mathcal{A} outputs (suppressing unnecessary terms) as the signature forgery.

- Case 2: \mathcal{A} outputs a signature σ that uses a different pk_{TC} as the SGX functionality \mathcal{F}_{sgx} . In this case, \mathcal{B} will seek to break Σ_{sgx} . \mathcal{B} interacts with a signature challenger Ch, who generates some $(\text{pk}^*, \text{sk}^*)$ pair, and gives to \mathcal{B} the public key pk^* . \mathcal{B} simulates \mathcal{F}_{sgx} by implicitly setting $\text{pk}_{\text{sgx}} := \text{pk}^*$. Whenever \mathcal{F}_{sgx} needs to make a signature with sk_{sgx} , \mathcal{B} simply passes the signature query onto Ch. In this case, in order for \mathcal{A} to succeed, it must produce a valid signature σ_{att} for a different public key pk' . Therefore, \mathcal{B} simply outputs this as a signature forgery.

□

Lemma A.1. \mathcal{C}_{TC} will never attempt to send money in **Deliver** or **Cancel** that was not deposited with the given id.

Proof. First we note that there are only three lines on which \mathcal{C}_{TC} sends money: (2), (3), and (5). Second, for a request id, $\$f$ is deposited. Third, because $\text{isCanceled}[\text{id}]$ is only set immediately prior to line (5) and line (2) is only reachable if $\text{isCanceled}[\text{id}]$ is set, it is impossible to reach line (2) without reaching line (5).

We now consider cases based on which of lines (3) and (5) are reached first (since at least one must be reached to send any money).

- *Line (5) is reached first.* In this case, line (5) sends $\$f - \G_{\emptyset} and allows $\$G_{\emptyset}$ to remain. Future calls to **Cancel** with this id will fail the $\text{isCanceled}[\text{id}]$ not check assertion, so line (5) can never be reached again with this id. If \mathcal{W}_{TC} invokes **Deliver** after this point, the first such invocation will satisfy the predicate on line (1) and proceed to set $\text{isDelivered}[\text{id}]$ and reach line (2). Any future entries to **Deliver** with id will fail to satisfy the predicate on line (1) and then fail an assertion and abort prior to line (3). Since line (2) sends $\$G_{\emptyset}$, the total money sent in connection with id is $(\$f - \$G_{\emptyset}) + \$G_{\emptyset} = \f .
- *Line (3) is reached first.* In this case, line (3) send the full $\$f$ immediately after setting $\text{isDelivered}[\text{id}]$. With $\text{isDelivered}[\text{id}]$ set, any call to **Cancel** with id will fail an assertion prior to line (5) and any future call to **Deliver** with id will fail to satisfy the predicate on line (1) and also fail an assertion prior to reaching line (3). Thus no further money will be distributed in connection with id.

□

Theorem 2.2 (Gas Sustainability). *Town Crier is $\$G_{\max}$ -gas sustainable.*

Proof. By assumption, \mathcal{W}_{TC} is seeded with at least $\$G_{\max}$ money. Thus it suffices to prove that, given an honest Relay, \mathcal{W}_{TC} will have at least as much money after invoking **Deliver** as it did before.

An honest Relay will never ask for a response for the same id more than once. **Deliver** only responds to messages from \mathcal{W}_{TC} , and `isDelivered[id]` is only set inside **Deliver**, so therefore we know that `isDelivered[id]` is not set for this id. We now consider the case where `isCanceled[id]` is set upon invocation of **Deliver** and the case where it is not.

- `isCanceled[id]` *not set*: In this case the predicate on line (1) of the protocol returns false. Because the Relay is honest, id exists and `params = params'`. The enclave always provides $\$g_{\text{dvr}} = \G_{\max} (which it has by assumption) and **Request** ensures that $\$f \leq \G_{\max} . Thus, coupled with the knowledge that `isDeliver[id]` is not set, all assertions pass and we progress through lines (3) and (4). Now we must show that at line (3) \mathcal{C}_{TC} had $\$f$ to send and that the total gas spend to execute **Deliver** does not exceed $\$f$.

To see that \mathcal{C}_{TC} had sufficient funds, we note that upon entry to **Deliver**, both `isDelivered[id]` and `isCanceled[id]` must have been unset. The first we showed above. The second is because, given the first, if `isCanceled[id]` were set, the predicate on line (1) would have returned true, sending execution on a path that would not encounter (4). This means that line (5) was never reached because the preceding line sets `isCanceled[id]`. Because (2), (3), and (5) are the only lines that remove money from \mathcal{C}_{TC} and $\$f$ was deposited as part of **Request**, it must be the case that $\$f$ is still in the contract.

To see how much gas is spent, we first note that $\$G_{\min}$ is defined to be the amount of gas needed to execute **Deliver** along this execution path not

including line (4). Since $\$g_{\text{clbk}}$ is set to $\$f - \G_{min} and line (4) is limited to using $\$g_{\text{clbk}}$ gas, the total gas spent on this execution of **Deliver** is at most $\$G_{\text{min}} + (\$f - \$G_{\text{min}}) = \f .

- *isCanceled[id] is set*: Here the predicate on line (1) returns true. Along this execution path \mathcal{C}_{TC} sends \mathcal{W}_{TC} $\$G_{\emptyset}$ and quickly returns. $\$G_{\emptyset}$ is defined as the amount of gas necessary to execute this execution path, so we need only show that \mathcal{C}_{TC} has $\$G_{\emptyset}$ available to send.

Because *isCanceled[id]* is set, it must be the case that **Cancel** was invoked with *id* and reached line (5). Gas exhaustion in **Cancel** is not a concern because it would abort and revert the entire invocation. This is only possible if the data retrieval and all assertions in **Cancel** succeed. In particular, this means that *id* corresponds to a valid request which deposited $\$f$. Line (5) returns $\$f - \G_{\emptyset} to \mathcal{C}_U , but it leaves $\$G_{\text{min}}$ from the original $\$f$. Moreover, if **Cancel** is invoked multiple times with the same *id*, all but the first will fail due to the assert that *isCanceled[id]* is not set and the fact that any invocation that reaches (5) will set *isCanceled* for that *id*. Since only lines (2), (3), and (5) can remove money from \mathcal{C}_{TC} and line (3) will never be called in this case, there will still be exactly $\$G_{\text{min}}$ available when this invocation of **Deliver** reaches line (2).

□

Theorem 2.3 (Fair Expenditure for Honest Requester). *For any params and callback, let $\$G_{\text{req}}$ and $\$F$ be the respective values chosen by an honest requester for $\$g_{\text{req}}$ and $\$f$ when submitting the request $(\text{params}, \text{callback}, \$f, \$g_{\text{req}})$. For any such request submitted by an honest user \mathcal{C}_U , one of the following holds:*

- *callback is invoked with a valid datagram matching the request parameters params*

and the requester spends at most $\$G_{req} + \$G_{cncl} + \$F$.

- The requester spends at most $\$G_{req} + \$G_{cncl} + \$G_{\emptyset}$.

Proof. C_U is honest, so she will first spend $\$G_{req}$ to invoke **Request**(params, callback, $\$F$). Ethereum does not allow money to change hands without the payer explicitly sending money. Therefore we must only examine the explicit function invocations and monetary transfers initiated by C_U in connection with the request. It is impossible for C_U to lose more money than she gives up in these transactions even if TC is malicious.

- *Request Delivered:* If protocol line (4) is reached, then we are guaranteed that $\text{params} = \text{params}'$ and $\$g_{dvr} \geq \F . By Theorem 2.1, the datagram must therefore be authentic for params. Because $\$F$ is chosen honestly for callback, $\$F - \G_{min} is enough gas to execute callback, so callback will be invoked with a datagram that is a valid and matches the request parameters.

In this case, the honest requester will have spent $\$G_{req}$ to invoke **Request** and $\$F$ in paying TC's cost for **Deliver**. The requester may have also invoked **Cancel** at most once at the cost of $\$G_{cncl}$. While C_U may not receive any refund due to **Cancel** aborting, C_U will still have spent at most $\$G_{req} + \$G_{cncl} + \$F$.

- *Request not Delivered:* The request not being delivered means that line (4) is never reached. This can only happen if **Deliver** is never called with a valid response or if `isCanceled[id]` is set before `deliver` is called. The only way to set `isCanceled[id]` is for C_U to invoke **Cancel** with `isDelivered[id]` not set. If `deliver` is not executed, we assume that an honest requester will eventually invoke **Cancel**, so this case will always reach line (5). When line (5) is

reached, then \mathcal{C}_U will have spent $\$G_{\text{req}} + \F while executing **Request**, and spent $\$G_{\text{cncl}}$ in **Cancel** and will attempt to retrieve $\$F - \G_{\emptyset} .

The retrieval will succeed because \mathcal{C}_{TC} will always have the funds to send \mathcal{C}_U $\$F - \G_{\emptyset} . To see this, Lemma A.1 allows us to consider only **Deliver** and **Cancel** calls associated with `id`.

Since line (5) is reached, it must be the case the `isDelivered[id]` is not set. This means that neither lines (2) nor (3) were reached since the line before each sets `isDelivered[id]`. The lines preceding those two and (5) are the only lines that remove money from the contract. Line (5) cannot have been reached before because \mathcal{C}_U is assumed to be honest, so she will not invoke **Cancel** twice for the same request and if any other user invokes **Cancel** for this request, the $\mathcal{C}_U = \mathcal{C}'_U$ assertion will fail and the invocation will abort before line (5). Because none of (2), (3), or (5) has been reached before and \mathcal{C}_U deposited $\$F > \$G_{\text{min}} > \$G_{\emptyset}$ on **Request**, it must be the case that \mathcal{C}_{TC} has $\$F - \G_{\emptyset} left.

This means the total expenditure in this case will be

$$\begin{aligned} & \$G_{\text{req}} + \$G_{\text{cncl}} + \$F - (\$F - \$G_{\emptyset}) \\ & = \$G_{\text{req}} + \$G_{\text{cncl}} + \$G_{\emptyset}. \end{aligned}$$

□

A.3 Applications and Code Samples

We now elaborate on the applications described in Section 2.9.1 and we show a short Solidity code sample for one of these applications, to demonstrate first-

hand what a requester contract would look like to call Town Crier’s authenticated data feed service.

Financial derivative (CashSettledPut). Financial derivatives are among the most commonly cited smart contract applications, and exemplify the need for a data feed on financial instruments. We implemented an example contract `CashSettledPut` for a *cash-settled put option*. This is an agreement for one party to buy an asset from the other at an agreed upon price on or before a particular date. It is “cash-settled” in that the sale is implicit, i.e., no asset changes hands, only cash reflecting the asset’s value. In our implementation, the issuer of the option specifies a strike price P_S , expiration date, unit price P_U , and maximum number of units M she is willing to sell. A customer may send a request to the contract specifying the number X of option units to be purchased and containing the associated fee ($X \cdot P_U$). A customer may then exercise the option by sending another request prior to the expiration date. `CashSettledPut` calls TC to retrieve the closing price P_C of the underlying instrument on the day the option was exercised, and pays the customer $X \cdot (P_S - P_C)$. To ensure sufficient funding to pay out, the contract must be endowed with ether value at least $M \cdot P_S$.

In Figure A.1 we describe the protocol for `CashSettledPut`. We omit the full source code due to length and complexity.

Flight insurance (FlightIns). Flight insurance indemnifies a purchaser should her flight be delayed or canceled. We have implemented a simple flight insurance contract called `FlightIns`. Our implementation showcases TC’s *private-datagram* feature to address an obvious concern: customers may not wish to reveal their travel plans publicly on the blockchain.

An insurer stands up FlightIns with a specified policy fee, payout, and lead time ΔT . (ΔT is set large enough to ensure that a customer can't anticipate flight cancellation or delay due to weather, etc.) To purchase a policy, a customer sends the FlightIns a ciphertext C under the TC's public key pk_{TC} of the ICAO flight number FN and scheduled time of departure T_D for her flight, along with the policy fee. FlightIns sends TC a private-datagram request containing the current time T and the ciphertext C . TC decrypts C and checks that the lead time meets the policy requirement, i.e., that $T_D - T \geq \Delta T$. TC then scrapes a flight information data source several hours after T_D to check the flight status, and returns to FlightIns predicates on whether the lead time was valid and whether the flight has been delayed or canceled. If both predicates are true, then FlightIns returns the payout to the customer. Note that FN is never exposed in the clear.

Despite the use of private datagrams, FlightIns as described here still poses a privacy risk, as the *timing* of the predicate delivery by TC leaks information about T_D , which may be sensitive information; this, and the fact that the payout is publicly visible, could also indirectly reveal FN . FlightIns addresses this issue by including in the private datagram request another parameter $t > T_D$ specifying the time at which predicates should be returned. By randomizing t and making $t - T_D$ sufficiently large, FlightIns can substantially reduce the leakage of timing information.

In Fig. A.2 we include a full implementation of FlightIns in Solidity.

Steam Marketplace (SteamTrade). Steam [22] is an online gaming platform that supports thousands of games and maintains its own marketplace, where users can trade, buy, and sell games and other virtual items. We implement a contract

for the sale of games and items for ether that showcases TC's support for custom datagrams through the use of Steam's access-controlled API.

A user intending to sell items creates a contract SteamTrade with her Steam account number ID_S , a list L of items for sale, a price P , and a ciphertext C under the TC's public key pk_{TC} of her Steam API key. In order to purchase the list of items, a buyer first uses a Steam client to create a trade offer requesting each item in L . The buyer then submits to SteamTrade her Steam account number ID_U , a length of time T_U indicating how long the seller has to respond to the offer, and an amount of ether equivalent to the price P . SteamTrade sends TC a custom datagram containing the current time T , ID_U , T_U , L , and the encrypted API key C . TC decrypts C to obtain the API key, delays for time T_U , then retrieves all trades between the two accounts using the provided API key within that time period. TC verifies whether or not a trade exactly matching the items in L successfully occurred between the two accounts and returns the result to SteamTrade. If such a trade occurred, SteamTrade sends the buyer's ether to the seller's account. Otherwise the buyer's ether is refunded.

In Fig. A.3 we describe the protocol for SteamTrade. We again omit the full source code due to length and complexity.

CashSettledPut blockchain contract

Constants

T_{stock} := Town Crier stock info request type
 $\$F_{\text{TC}}$:= fee payed to TC for datagram delivery

Functions

Init: On rcv (\mathcal{C}_{TC} , ticker, P_S , P_U , M , expr, $\$f$) from $\mathcal{W}_{\text{issuer}}$
Assert $\$f = (P_S - P_U) \cdot M + \F_{TC}
Save all inputs and $\mathcal{W}_{\text{issuer}}$ to storage.

Buy: On rcv (X , $\$f$) from \mathcal{W}_U :
Assert isRecovered not set
and timestamp < expr
and $\mathcal{W}_{\text{buyer}}$ not set
and $X \leq M$
and $\$f = (X \cdot P_U)$
Set $\mathcal{W}_{\text{buyer}} = \mathcal{W}_U$
Save X to storage
Send $(P_S - P_U)(M - X)$ to $\mathcal{W}_{\text{issuer}}$
// Hold $P_S \cdot X + \$F_{\text{TC}}$

Put: On rcv () from $\mathcal{W}_{\text{buyer}}$:
and timestamp < expr
and isPut not set
Set isPut
params := [T_{stock} , ticker]
callback := this.**Settle**
 \mathcal{C}_{TC} .**Request**(params, callback, $\$F_{\text{TC}}$)

Settle: On rcv (id, P) from \mathcal{C}_{TC} :
If $P \geq P_S$
Send $P_S \cdot X$ to $\mathcal{W}_{\text{issuer}}$
Return
Send $(P_S - P)X$ to $\mathcal{W}_{\text{buyer}}$
Send all money in contract to $\mathcal{W}_{\text{issuer}}$
Send $P \cdot X$ to $\mathcal{W}_{\text{issuer}}$

Recover: On rcv () from $\mathcal{W}_{\text{issuer}}$:
and isPut not set
and isRecovered not set
and ($\mathcal{W}_{\text{buyer}}$ not set
or timestamp \geq expr)
Set isRecovered
Send all money in contract to $\mathcal{W}_{\text{issuer}}$

Figure A.1: The CashSettledPut application contract.

```

// A simple flight insurance contract using Town Crier's private datagram.
contract FlightIns {
    uint    constant TC_REQ_TYPE = 0;
    uint    constant TC_FEE      = (35000 + 20000) * 5 * 10**10;
    uint    constant FEE        = 10**18;      // $5 in wei
    uint    constant PAYOUT     = 2 * 10**19;  // $200 in wei
    uint32  constant MIN_DELAY  = 30;

    // The function identifier in Solidity is the first 4 bytes
    // of the sha3 hash of the functions' canonical signature.
    // This contract's callback is bytes4(sha3("pay(uint64,bytes32)"))
    bytes4  constant CALLBACK_FID = 0x3d622256;

    TownCrier tc;
    address[2**64] requesters;

    // Constructor which sets the address of the Town Crier contract.
    function FlightIns(TownCrier _tc) public {
        tc = _tc;
    }

    // A user can purchase insurance through this entry point.
    // encFN is an encryption of the flight number and date
    // as well as the time when Town Crier should respond to the request.
    function insure(bytes32[] encFN) public {
        if (msg.value != FEE) return;

        // Adding money to a function call involves calling ".value()"
        // on the function itself before calling it with arguments.
        uint64 requestId =
            tc.request.value(TC_FEE)(TC_REQ_TYPE, this, CALLBACK_FID, encFN);
        requesters[requestId] = msg.sender;
    }

    // This is the entry point for Town Crier to respond to a request.
    function pay(uint64 requestId, bytes32 delay) public {
        // Check that this is a response from Town Crier
        // and that the ID is valid and unfulfilled.
        address requester = requesters[requestId];
        if (msg.sender != address(tc) && requester == 0) return;

        if (uint(delay) >= MIN_DELAY) {
            address(requester).send(PAYOUT);
        }
        requesters[requestId] = 0;
    }
}

```

Figure A.2: Solidity code for the FlightIns application contract.

SteamTrade **blockchain contract**

Constants

T_{steam} := Town Crier Steam trade request type
 $\$F_{\text{TC}}$:= fee payed to TC for datagram delivery

Functions

Init: On recv ($C_{\text{TC}}, ID_S, encAPI_S, List_I, P$) from \mathcal{W}_S :
Save all inputs and \mathcal{W}_S to storage.

Buy: On recv ($ID_U, T_U, \$f$) from \mathcal{W}_U :
Assert $\$f = P$
params := [$encAPI_S, ID_U, T_U, List_I$]
callback := this.**Pay**
id := C_{TC} .**Request**(params, callback, $\$F_{\text{TC}}$)
Store (id, \mathcal{W}_U)

Pay: On recv (id, *status*) from C_{TC} :
Retrieve and remove stored (id, \mathcal{W}_U)
// Abort if not found
If *status* > 0
Send $\$F_{\text{price}}$ to $\mathcal{W}_{\text{seller}}$
Else
send $\$F_{\text{price}}$ to \mathcal{W}_U

Figure A.3: The FlightIns application contract.

APPENDIX B

DECO

B.1 Protocols details for GCM

B.1.1 Preliminaries

GCM is an authenticated encryption with additional data (AEAD) cipher. To encrypt, the GCM cipher takes as inputs a tuple (k, IV, M, A) : a secret key, an initial vector, a plaintext of multiple AES blocks, and additional data to be included in the integrity protection; it outputs a ciphertext C and a tag T . Decryption reverses the process. The decryption cipher takes as input (k, IV, C, A, T) and first checks the integrity of the ciphertext by comparing a recomputed tag with T , then outputs the plaintext.

The ciphertext is computed in the counter mode: $C_i = \text{AES}(k, \text{inc}^i(IV)) \oplus M_i$ where inc^i denotes incrementing IV for i times (the exact format of inc is immaterial.)

The tag $\text{Tag}(k, IV, C, A)$ is computed as follows. Given a vector $\mathbf{X} \in \mathbb{F}_{2^{128}}^m$, the associated GHASH polynomial $P_{\mathbf{X}} : \mathbb{F}_{2^{128}} \rightarrow \mathbb{F}_{2^{128}}$ is defined as $P_{\mathbf{X}}(h) = \sum_{i=1}^m X_i \cdot h^{m-i+1}$ with addition and multiplication done in $\mathbb{F}_{2^{128}}$. Without loss of generality, suppose A and C are properly padded. Let ℓ_A and ℓ_C denote their length. A GCM tag is

$$\text{Tag}(k, IV, C, A) := \text{AES}(k, IV) \oplus P_{A\|C\|\ell_A\|\ell_C}(h) \quad (\text{B.1})$$

where $h = \text{AES}(k, \mathbf{0})$.

When GCM is used in TLS, each plaintext record D is encrypted as follows. A unique nonce n is chosen and the additional data κ is computed as a concatenation of the sequence number, version, and length of D . GCM encryption is invoked to generate the payload record as $M = n \parallel \text{GCM}(k, n, D, \kappa)$. We refer readers to [110] for a complete specification.

B.1.2 Query execution

The 2PC protocols for verifying tags and decrypting records are specified in Fig. B.1.

Tag creation/verification. Computing or verifying a GCM tag involves evaluating Eq. (B.1) in 2PC. A challenge is that Eq. (B.1) involves both arithmetic computation (e.g., polynomial evaluation in $\mathbb{F}_{2^{128}}$) as well as binary computation (e.g., AES). Performing multiplication in a large field in a binary circuit is expensive, while computing AES (defined in $\text{GF}(2^8)$) in $\mathbb{F}_{2^{128}}$ incurs high overhead. Even if the computation could somehow be separated into two circuits, evaluating the polynomial alone—which takes approximately 1,000 multiplications in $\mathbb{F}_{2^{128}}$ for *each* record—would be unduly expensive.

Our protocol removes the need for polynomial evaluation. The actual 2PC protocol involves only binary operations and thus can be done in a single circuit. Moreover, the per-record computation is reduced to only one invocation of 2PC-AES.

The idea is to compute shares of $\{h^i\}$ (in a 2PC protocol) in a preprocessing phase at the beginning of a session. The overhead of preprocessing is amortized

over the session because the same h used for all records that follow. With shares of $\{h^i\}$, \mathcal{P} and \mathcal{V} can compute shares of a polynomial evaluation $P_{A\|C\|\ell_A\|\ell_C}(h)$ locally. They also compute $\text{AES}(k, IV)$ in 2PC to get a share of $\text{Tag}(k, IV, C, A)$. In total, only one invocation of 2PC-AES is needed to check the tag for each record.

It is critical that \mathcal{V} never responds to the same IV more than once; otherwise \mathcal{P} would learn h . Specifically, in each response, \mathcal{V} reveals a blinded linear combination of her shares $\{h_{\mathcal{V},i}\}$ in the form of $\mathcal{L}_{IV,X} = \text{AES}(k, IV) \oplus \sum_i X_i \cdot h_{\mathcal{V},i}$. It is important that the value is blinded by $\text{AES}(k, IV)$ because a single unblinded linear combination of $\{h_{\mathcal{V},i}\}$ would allow \mathcal{P} to solve for h . Therefore, if \mathcal{V} responds to the same IV twice, the blinding can be removed by adding the two responses (in $\mathbb{F}_{2^{128}}$): $\mathcal{L}_{IV,X} \oplus \mathcal{L}_{IV,X'} = \sum_i (X_i + X'_i) \cdot h_{\mathcal{V},i}$. This follows from the nonce uniqueness requirement of GCM [220].

Encrypting/decrypting records. Once tags are properly checked, decryption of records is straightforward. \mathcal{P} and \mathcal{V} simply compute AES encryption of $\text{inc}^i(IV)$ with 2PC-AES. A subtlety to note is that \mathcal{V} must check that the counters to be encrypted have *not* been used as IV previously. Otherwise \mathcal{P} would learn h to \mathcal{P} in a manner like that outlined above.

B.1.3 Proof Generation

Revealing a block. \mathcal{P} wants to convince \mathcal{V} that an AES block B_i is the i th block in the encrypted record $\hat{\text{réc}}$. The proof strategy is as follows: 1) prove that AES block B_i encrypts to the ciphertext block \hat{B}_i and 2) prove that the tag is correct.

Proving the correct encryption requires only 1 AES in ZKP. Naïvely done, proving the correct tag incurs evaluating the GHASH polynomial of degree 512 and 2 AES block encryptions in ZKP.

We manage to achieve a much more efficient proof by allowing \mathcal{P} to reveal two encrypted messages $\text{AES}(k, IV)$ and $\text{AES}(k, 0)$ to \mathcal{V} , thus allowing \mathcal{V} to verify the tag (see Eq. (B.1)). \mathcal{P} only needs to prove the correctness of encryption in ZK and that the key used corresponds to the commitment, requiring 2 AES and 1 SHA-2 (\mathcal{P} commits to $k_{\mathcal{P}}$ by revealing a hash of the key). Thus, the total cost is 3 AES and 1 SHA-2 in ZKP.

Revealing a TLS record. The proof techniques are a simple extension from the above case. \mathcal{P} reveals the entire record rec and proves correct AES encryption of all the AES blocks, resulting in a total 514 AES and 1 SHA-2 in ZKP.

Revealing a TLS record except for a block. Similar to the above case, \mathcal{P} proves encryption of all the blocks in the record except one, resulting in a total 513 AES and 1 SHA-2 in ZKP.

B.2 Security proofs

Recall Theorem 3.1. We now prove that the protocol in Fig. 3.7 securely realizes $\mathcal{F}_{\text{Oracle}}$. Specifically, we show that for any real-world adversary \mathcal{A} , we can construct an ideal world simulator Sim , such that for all environments \mathcal{Z} , the ideal execution with Sim is indistinguishable from the real execution with \mathcal{A} . We refer readers to [198, 70] for simulation-based proof techniques.

Proof. Recall that we assume \mathcal{S} is honest throughout the protocol. Hence, we only consider cases where \mathcal{A} maliciously corrupts either \mathcal{P} or \mathcal{V} . This means that we only need to construct ideal-world simulators for the views of \mathcal{P} and \mathcal{V} .

Malicious \mathcal{P} . We wish to show the prover-integrity guarantee. Basically, if \mathcal{V} receives (b, \mathcal{S}) , then \mathcal{P} must have input some θ_s such that $\mathcal{S}(\text{Query}(\theta_s)) = R$ and $b = \text{Stmt}(R)$.

Given a real-world PPT adversary \mathcal{A} , Sim proceeds as follows:

1. Sim runs \mathcal{A} , \mathcal{F}_{ZK} and \mathcal{F}_{2PC} internally. Sim forwards any input z from \mathcal{Z} to \mathcal{A} and records the traffic going to and from \mathcal{A} .
2. Upon request from \mathcal{A} , Sim runs 3P-HS as \mathcal{V} (using \mathcal{F}_{2PC} as a sub-routine). During 3P-HS, when \mathcal{A} outputs a message m intended for \mathcal{S} , Sim forwards it to $\mathcal{F}_{\text{Oracle}}$ as $(\text{sid}, \mathcal{S}, m)$ and forwards (sid, m) to \mathcal{A} if it receives any messages from $\mathcal{F}_{\text{Oracle}}$. By the end, Sim learns $Y_{\mathcal{P}, s_{\mathcal{V}}}, k_{\mathcal{V}}^{\text{MAC}}$.
3. Upon request from \mathcal{A} , Sim runs 2PC-HMAC as \mathcal{V} , using $k_{\mathcal{V}}^{\text{MAC}}$ as input. Again, Sim uses \mathcal{F}_{2PC} as a sub-routine to run 2PC-HMAC and forwards messages to \mathcal{S} as above and forwards the response from \mathcal{S} to \mathcal{A} . Sim records the messages between \mathcal{A} and \mathcal{S} during this stage in (\hat{Q}, \hat{R}) . Note that these are ciphertext records.
4. When \mathcal{A} sends $(\text{sid}, \hat{Q}, \hat{R}, k_{\mathcal{P}}^{\text{MAC}})$, reply with $(\text{sid}, k_{\mathcal{V}}^{\text{MAC}})$.
5. Upon receiving $(\text{sid}, \text{"prove"}, x, w)$ (with $x = (k^{\text{Enc}}, \theta_s, Q, R)$ and $w = (\hat{Q}, \hat{R}, k^{\text{MAC}}, b)$) from \mathcal{A} , Sim checks that $\hat{Q} = \text{CBC_HMAC}(k^{\text{Enc}}, k^{\text{MAC}}, Q)$, $\hat{R} = \text{CBC_HMAC}(k^{\text{Enc}}, k^{\text{MAC}}, R)$, and that $\text{Query}(\theta_s) = Q$.

6. If all of the above checks passed, Sim sends θ_s to $\mathcal{F}_{\text{Oracle}}$ and instructs $\mathcal{F}_{\text{Oracle}}$ to send the output to \mathcal{V} . Sim outputs whatever \mathcal{A} outputs.

Now we argue that the ideal execution with Sim is indistinguishable from the real execution with \mathcal{A} .

Hybrid H_1 is the real-world execution of $\text{Prot}_{\text{DECO}}$.

Hybrid H_2 is the same as H_1 , except that Sim simulates \mathcal{A} , \mathcal{F}_{ZK} and \mathcal{F}_{2PC} internally. Sim records and forwards its private θ_s input to \mathcal{A} . For each step of $\text{Prot}_{\text{DECO}}$, Sim forwards all messages between \mathcal{A} and \mathcal{V} and \mathcal{A} and \mathcal{S} , as in the real execution. Since the simulation of ideal functionality is perfect, H_1 and H_2 are indistinguishable.

Hybrid H_3 is the same as H_2 , except that \mathcal{V} sends input to $\mathcal{F}_{\text{Oracle}}$, which sends it to Sim and Sim simulates \mathcal{V} internally. Specifically, Sim samples \hat{s}_V and uses $\hat{s}_V \cdot Y$ to derive a share of the MAC key \hat{K} , which it uses in the sequential 2PC-HMAC invocations. Upon receiving $(\text{sid}, \hat{Q}, \hat{R}, k_P^{\text{MAC}})$, Sim sends $(\text{sid}, k_V^{\text{MAC}})$ to \mathcal{A} . If Sim receives $(\text{sid}, \text{"prove"}, x, w)$, it internally forwards it to \mathcal{F}_{ZK} , verifies its output as \mathcal{V} and also, sends θ_s to $\mathcal{F}_{\text{Oracle}}$. The indistinguishability between H_2 and H_3 is immediate because \hat{s}_V is uniformly random.

Hybrid H_4 is the same as H_3 , except Sim adds the checks in Step 5. The indistinguishability between H_3 and H_4 can be shown by checking that if any of the checks fails, \mathcal{V} would abort the real-world execution as well. There are two reasons that Sim may abort: 1) Q, R from \mathcal{A} is not originally from \mathcal{S} , or 2) $k^{\text{Enc}}, k^{\text{MAC}}$ from \mathcal{A} is not the same key as derived during the handshake. We now show that both conditions would trigger \mathcal{V} to abort in H_3 as well except with negligible probability.

- Assuming DL is hard in the group used in the handshake, \mathcal{A} cannot learn \hat{s}_V . Furthermore, due to the security of 2PC, \mathcal{A} cannot learn the session MAC key k^{MAC} . If \mathcal{A} maliciously selects \hat{Y}_P correlated with \hat{Y}_V , it would have to find the discrete log of $\hat{Y}_P - Y_V$, denoted \hat{s}_P . Without such a \hat{s}_P , except with negligible probability, the output shares \hat{K}_V^{MAC} and \hat{K}_P^{MAC} of 3P-HS would fail to verify a MAC from an honest server whose MAC key is derived using \hat{Y}_P in 2PC-HMAC, later in the protocol.
- The unforgeability guarantee of HMAC ensures that without knowledge of k^{MAC} , \mathcal{A} cannot forge tags that verifies against k^{MAC} (checked by \mathcal{V} in the last step of $\text{Prot}_{\text{DECO}}$).
- If \mathcal{A} sends a different $(k^{\text{Enc}}, k^{\text{MAC}})$ pair than that derived during the handshake to Sim and the decryption and MAC check succeeds, then \mathcal{A} would have broken the receiver-binding property of CBC-HMAC [139].

It remains to show that H_4 is exactly the same the ideal execution. Due to Step 5 and 6, $\mathcal{F}_{\text{Oracle}}$ delivers $(\text{sid}, \text{Stmt}(R), \mathcal{S})$ to \mathcal{V} only if $\exists \theta_s$ from \mathcal{A} such that R is the response from \mathcal{S} to $\text{Query}(\theta_s)$.

Malicious \mathcal{V} . As the verifier is corrupt, we are interested in showing the verifier-integrity and privacy guarantees. Sim proceeds as follows:

1. Sim runs \mathcal{A} , \mathcal{F}_{ZK} and \mathcal{F}_{2PC} internally to simulate the real-world interaction with the prover \mathcal{P} . Given input z from the environment \mathcal{Z} , Sim forwards it to \mathcal{A} .
2. Upon receipt of Query and Stmt from \mathcal{A} , forward them to $\mathcal{F}_{\text{Oracle}}$ and instruct it to send them to \mathcal{P} .

3. After \mathcal{P} sends θ_s to $\mathcal{F}_{\text{Oracle}}$, $\mathcal{F}_{\text{Oracle}}$ sends the output (sid, Q, R) to \mathcal{P} . Sim gets $(\text{sid}, \text{Stmt}(R), \mathcal{S})$ from $\mathcal{F}_{\text{Oracle}}$ and learns the record sizes $|Q|, |R|$.
4. Send $(\text{sid}, \mathcal{S}, \text{handshake})$ to $\mathcal{F}_{\text{Oracle}}$, where handshake contains client handshake messages and receive certificate and signatures of \mathcal{S} from $\mathcal{F}_{\text{Oracle}}$. Note that at the end of the server handshake, \mathcal{P} receives and sends finished messages, which we denote “serverFinished” and “proverFinished”. The finished messages include HMAC tags, which we denote $\tau_{\mathcal{S}}$ and $\tau_{\mathcal{P}}$ (tags on \mathcal{S} and \mathcal{P} ’s messages respectively).
5. Upon request from \mathcal{A} , Sim runs 3P-HS as \mathcal{P} , using the server handshake messages received in the previous step, learning $s_{\mathcal{P}}, Y_V, k^{\text{Enc}}, k_{\mathcal{P}}^{\text{MAC}}$.
6. Sim starts 2PC-HMAC as \mathcal{P} to compute a tag τ_q on a random $Q' \leftarrow_{\$} \{0, 1\}^{|Q|}$.
7. Sim uses a random key \hat{k} to compute a tag τ_r on a random $R' \leftarrow_{\$} \{0, 1\}^{|R|}$.
8. Let $\hat{Q} = \text{CBC}(k^{\text{Enc}}, Q' \parallel \tau_q)$ and $\hat{R} = \text{CBC}(k^{\text{Enc}}, R' \parallel \tau_r)$. At the commit phase, Sim sends encrypted data $(\text{sid}, \hat{Q}, \hat{R}, k_{\mathcal{P}}^{\text{MAC}})$ to \mathcal{A} and receives k_V^{MAC} from \mathcal{A} .
9. Sim asserts that the handshake tag $\tau_{\mathcal{S}} = \text{HMAC}(k^{\text{MAC}}, \text{“serverFinished”})$ and that $\tau_{\mathcal{P}} = \text{HMAC}(k^{\text{MAC}}, \text{“proverFinished”})$.
10. Sim asserts that $\tau_q = \text{HMAC}(k^{\text{MAC}}, Q')$.
11. To simulate the appropriate delay, Sim also runs a dummy computation $\text{HMAC}(k^{\text{MAC}}, R')$ in parallel with Step 9.
12. Sim sends $(\text{sid}, \text{“proof”}, 1, (\hat{Q}, \hat{R}, k^{\text{MAC}}, \text{Stmt}(R)))$ to \mathcal{A} and outputs whatever \mathcal{A} outputs.

We argue that the ideal execution with Sim is indistinguishable from the real execution with \mathcal{A} in a series of hybrid worlds.

Hybrid H_1 is the real-world execution of $\text{Prot}_{\text{DECO}}$.

Hybrid H_2 is the same as H_1 , except that Sim simulates \mathcal{F}_{ZK} and \mathcal{F}_{2PC} internally. Sim also invokes \mathcal{F}_{Oracle} and gets $(sid, Stmt(R), \mathcal{S})$, learns record sizes $|Q|, |R|$. Since the simulation of ideal functionality is perfect, H_1 and H_2 is indistinguishable.

Hybrid H_3 is the same as H_2 , except that Sim simulates \mathcal{P} . Specifically, Sim samples s_P and uses $s_P \cdot Y$ to derive a share of the MAC key k_P^{MAC} . Then, Sim uses k_P^{MAC} and a random $Q' = \{0, 1\}^{|Q|}$ as inputs to 2PC-HMAC and receives the tag τ_q . Then, Sim uses a random key \hat{k} , and a random $R' = \{0, 1\}^{|R|}$ to compute a dummy tag τ_r . Afterwards, Sim commits, i.e., sends encryption of Q' and R' to \mathcal{A} . Sim also adds the checks in Step 9 and 10. To simulate the appropriate delay for checking a tag on R' , a plaintext of length $|R|$, Sim runs a dummy tag computation. Finally, Sim skips invoking \mathcal{F}_{ZK} and directly provides \mathcal{A} with the output obtained earlier from \mathcal{F}_{Oracle} , i.e., $Stmt(R)$, alongwith k^{MAC} , i.e. the tuple $(sid, \text{"proof"}, 1, (\hat{Q}, \hat{R}, k^{MAC}, Stmt(R)))$. \mathcal{A} cannot distinguish between the real and ideal executions because:

1. Since input sizes are equal, the number of invocations of 2PC-HMAC is also equal.
2. In each invocation of 2PC-HMAC and HMAC, \mathcal{A} learns one SHA-2 hash of the input message which is like a random oracle.
3. If the value of k_V^{MAC} provided by \mathcal{V} is correct, in both the real and ideal world, all tags should verify and the protocol should proceed to the next step and the time to run the checks should be indistinguishable from the real world.
4. \mathcal{A} can provide a malicious k_V^{MAC} in two ways:
 - Malicious k_V^{MAC} is provided by \mathcal{V} in Step 8: τ_S and τ_P will not verify in Step 9. Sim will then abort with the same delay as in the real world.

- \mathcal{A} inputs a malicious k_V^{MAC} to the 2PC-HMAC: τ_q will fail to verify in 10 by the same argument as in the malicious \mathcal{P} case.
5. Since $|Q'| = |Q|$ and $|R'| = |R|$, their encryptions are also of equal size and indistinguishable.
 6. In the end, \mathcal{A} receives the same output as the real execution.

□

B.3 Details on Key-Value Grammars and Two-Stage Parsing

B.3.1 Preliminaries and notation

We denote context-free grammars as $\mathcal{G} = (V, \Sigma, P, S)$ where V is a set of non-terminal symbols, Σ a set of terminal symbols, $P : V \rightarrow (V \cup \Sigma)^*$ a set of productions or rules and $S \in V$ the start-symbol. We define production rules for CFGs in standard notation using $'-'$ to denote a set minus and $'..'$ to denote a range. For a string w , a parser determines if $w \in \mathcal{G}$ by constructing a parse tree for w . The parse tree represents a sequence of production rules which can then be used to extract semantics.

B.3.2 Key-value grammars

These are grammars with the notion of key-value pairs. These grammars are particularly interesting for DECO since most API calls and responses are, in fact, key-value grammars.

Definition B.1. \mathcal{G} is said to be a key-value grammar if there exists a grammar \mathcal{H} , such that given any $s \in \mathcal{G}$, $s \in \mathcal{H}$, and \mathcal{H} can be defined by the following rules:

```
S → object
object → noPairsString open pair pairs close
pair → start key middle value end
pairs → pair pairs | ""
key → chars
value → chars | object
chars → char chars | ""
char → Unicode - escaped | escape escaped | addedChars
special → startSpecial | middleSpecial | endSpecial
start → unescapeds startSpecial
middle → unescapedm middleSpecial
end → unescapede endSpecial
escaped → special | escape | ...
```

In Definition B.1, S is the start non-terminal (represents a sentence in \mathcal{H}), the non-terminals `open` and `close` demarcate the opening and closing of the set of key-value pairs and `start`, `middle`, `end` are special strings demarcating the start of a key-value pair, separation between a key and a value and the end of the pair respectively.

In order to remove ambiguity in parsing special characters, i.e. characters which have special meaning in parsing a grammar, a special non-terminal, `escape` is used. For example, in JSON, keys are parsed when preceded by ‘whitespace double quotes’ (“) and succeeded by double quotes. If a key or value expression itself must contain double quotes, they must be preceded by a backslash (\), i.e.

escaped. In the above rules, the non-terminal unescaped before special characters means that they can be parsed as special characters. So, moving forward, we can assume that the production of a key-value pair is unambiguous. So, if a substring R' of a string R in the key-value grammar \mathcal{G} parses as a pair, R' must correspond to a pair in the parse tree of R .

Note that in Definition B.1, middle cannot derive an empty string, i.e. a non-empty string must mark middle to allow parsing keys from values. However, one of start and end can have an empty derivation, since they only demarcate the separation between value in one pair from key in the next. Finally, we note that in our discussion of two-stage parsing for key-value grammars, we only we consider permissible paths with the requirement that the selectively opened string, R_{open} corresponds to a pair.

B.3.3 Two-stage parsing for a locally unique key

Many key-value grammars enforce key uniqueness within a scope. For example, in JSON, it can be assumed that keys are unique within a JSON object, even though there might be duplicated keys across objects. The two-stage parsing for such grammars can be reduced to parsing a substring. Specifically, Trans extracts from R a continuous substring R' , such that the scope of a pair can be correctly determined, even within R' . For instance, in JSON, if $\text{cons}_{\mathcal{G}, \mathcal{G}'}(R, R')$ returns true iff R' is a prefix of R , then only parsing R' as a JSON, up to generating the sub-tree yielding R_{open} is sufficient for determining whether a string R_{open} corresponds to the correct context in R .

B.3.4 Grammars with unique keys

Given a key-value grammar \mathcal{G} we define a function which checks for uniqueness of keys, denoted $u_{\mathcal{G}}$. Given a string $s \in \mathcal{G}$ and another string k , $u_{\mathcal{G}}(s, k) = \text{true}$ iff there exists at most one substring of s that can be parsed as start k middle. Since $s \in \mathcal{G}$, this means, in any parse tree of s , there exists at most one branch with node key and derivation k . Let $\text{Parser}_{\mathcal{G}}$ be a function that returns true if its input is in the grammar \mathcal{G} . We say a grammar \mathcal{G} is a *key-value grammar with unique keys* if for all $s \in \mathcal{G}$ and all possible keys k , $u_{\mathcal{G}}(s, k) = \text{true}$, i.e. for all strings R, C :

$$\frac{\langle \text{Parser}_{\mathcal{G}}, R \rangle \Rightarrow \text{true}}{\langle u_{\mathcal{G}}, (R, C) \rangle \Rightarrow \text{true}}.$$

B.3.5 Concrete two-stage parsing for unique-key grammars

Let \mathcal{U} be a unique-key grammar as given above. We assume that \mathcal{U} is LL(1). This is the case for the grammars of interest in Section 3.6. See [140] for a general LL(1) parsing algorithm.

We instantiate a context function, $\text{CTX}_{\mathcal{U}}$ for a set T , such that T contains the permissible paths to a pair for strings in \mathcal{U} . We additionally allow $\text{CTX}_{\mathcal{U}}$ to take as input an auxiliary restriction, a key k (the specified key in \mathcal{P} 's output R_{open}). The tuple (T, k) is denoted S and $\text{CTX}_{\mathcal{U}}(S, \cdot, \cdot)$ as $\text{CTX}_{\mathcal{U}, S}$.

Let \mathcal{P} be a grammar given by the rule $S_{\mathcal{P}} \rightarrow \text{pair}$, where pair is the non-terminal in the production rules for \mathcal{U} and $S_{\mathcal{P}}$ is the start symbol in \mathcal{P} . We define $\text{Parser}_{\mathcal{P}, k}$ as a function that decides whether a string s is in \mathcal{P} and if so, whether the key in s equals k . On input R, R_{open} , $\text{CTX}_{\mathcal{U}, S}$ checks that: (a) R_{open} is a valid key-value pair with key k by running $\text{Parser}_{\mathcal{P}, k}$ (b) R_{open} parses as a key-value

pair in R by running an LL(1) parsing algorithm to parse R .

To avoid expensive computation of $\text{CTX}_{\mathcal{U},S}$ on a long string R , we introduce the transformation Trans , to extract the substring R' of R , such that $R' = R_{\text{open}}$ as per the requirements.

For string s, t , we also define functions $\text{substring}(s, t)$, that returns true if t is a substring of s and $\text{equal}(s, t)$ which returns true if $s = t$. We define $\text{cons}_{\mathcal{U},\mathcal{P}}$ with the rule:

$$\frac{\langle \text{substring}(R, R') \rangle \Rightarrow \text{true} \quad \langle \text{Parser}_{\mathcal{P},k}, R' \rangle \Rightarrow \text{true}}{\langle \text{cons}_{\mathcal{U},\mathcal{P}}, (R, R') \rangle \Rightarrow \text{true}}.$$

and $S' = \{S_{\mathcal{P}}\}$. Meaning, $\text{CTX}_{\mathcal{P}}(S, R', R_{\text{open}}) = \text{true}$ whenever $\text{equal}(R', R_{\text{open}})$ and the rule

$$\frac{\langle \text{equal}, (R', R_{\text{open}}) \rangle \Rightarrow \text{b}}{\langle \text{CTX}_{\mathcal{P}}, (S', R', R_{\text{open}}) \rangle \Rightarrow \text{b}}$$

holds for all strings R', R_{open} .

Claim B.1. $(\text{cons}_{\mathcal{U},\mathcal{P}}, S')$ are correct with respect to S .

Proof. We defer a formal proof and pseudocode for $\text{CTX}_{\mathcal{U},S}$ to a full version, but the intuition is that if R' is substring of R , a key-value pair R_{open} is parsed by $\text{Parser}_{\mathcal{P}}$, then the same pair must have been a substring of \mathcal{U} . Due to global uniqueness of keys in \mathcal{U} , there exists only one such pair R_{open} and $\text{CTX}_{\mathcal{U}}(S, R, R_{\text{open}})$ must be true. \square

Post-handshake protocols for GCM

Private input: $k_{\mathcal{P}}$ and $k_{\mathcal{V}}$ from \mathcal{P} and \mathcal{V} respectively

Protocol for preprocessing

On initialization: \mathcal{P} (and \mathcal{V}) sends $k_{\mathcal{P}}$ and $(k_{\mathcal{V}})$ to \mathcal{F}_{PP} and wait for output $\{h_{\mathcal{P},i}\}_i$ (and $\{h_{\mathcal{V},i}\}_i$).

\mathcal{F}_{PP} : After receiving k_1, k_2 from two parties, compute $h := \text{AES}(k_1 + k_2, \mathbf{0})$. Sample n random numbers $\{r_i\}_{i=1}^n$ and compute $\{h^i\}_{i=1}^n$ in $\mathbb{F}_{2^{128}}$. For $i \in [n]$, send r_i to player 1 and $r_i \oplus h^i$ to player 2.

Protocol for decrypting TLS records

Prover \mathcal{P} :

On receiving a record (IV, C, A, T) from \mathcal{S} :

- Let $X = A \| C \| \ell_A \| \ell_C$.
- Send $(k_{\mathcal{P}}, IV)$ to $\mathcal{F}_{\text{AES-EqM}}$ and wait for output $c_{\mathcal{P}}$.
- Send (IV, X) to \mathcal{V} and wait for the response P .
- Compute $T' = P + c_{\mathcal{P}} + \sum_i X_i \cdot h_{\mathcal{P},i}$ in $\mathbb{F}_{2^{128}}$.
- Abort if $T' \neq T$. Otherwise, compute K such that $K_i = \text{inc}^i(IV)$ for $i \in [\ell_C]$. Send $(IV, \ell_C, \text{Decrypt})$ to \mathcal{V} .
- Send $(k_{\mathcal{P}}, K)$ to $\mathcal{F}_{\text{AES-EqM-Asym}}$ as party 1 and wait for output K' .
- Decrypt the message as $M_i = K'_i \oplus C_i$.

Verifier \mathcal{V} :

On receiving (IV, X) from \mathcal{P} :

- If IV found in store, abort. Otherwise store IV and proceed.
- Send $(k_{\mathcal{V}}, IV)$ to $\mathcal{F}_{\text{AES-EqM}}$ and wait for output $c_{\mathcal{V}}$.
- Compute $P = c_{\mathcal{V}} + \sum_i X_i \cdot h_{\mathcal{V},i}$ in $\mathbb{F}_{2^{128}}$
- Send P to \mathcal{P} .

On receiving $(IV, n, \text{Decrypt})$ from \mathcal{P} :

- Compute K such that $K_i = \text{inc}^i(IV)$ for $i \in [n]$.
- Abort if any K_i is found in store (as previously used IVs.)
- Send $(k_{\mathcal{V}}, K)$ to $\mathcal{F}_{\text{AES-EqM-Asym}}$ as party 2.

$\mathcal{F}_{\text{AES-EqM}}$: Wait for input (k_i, m_i) from party i for $i \in \{1, 2\}$. Abort if $m_1 \neq m_2$. Sample $r \leftarrow_{\$} \mathbb{F}$. Compute $c = \text{AES}(k_1 \oplus k_2, m_1)$. Send r to party 1 and $c \oplus r$ to party 2.

$\mathcal{F}_{\text{AES-EqM-Asym}}$: Wait for input (k_i, m_i) from party i for $i \in \{1, 2\}$. Abort if $m_1 \neq m_2$. Compute $c = \text{AES}(k_1 \oplus k_2, m_1)$. Send c to party 1 and \perp to party 2.

Figure B.1: The post-handshake protocols for AES-GCM.

APPENDIX C

CHURP

C.1 Security Proof for Opt – CHURP

Recall that a protocol for dynamic-committee proactive secret sharing satisfies secrecy and integrity. We prove secrecy first.

Secrecy. We consider the handoff protocol of one epoch first. As described in Section 4.5.3, Opt-CHURP consists of three phases: Opt-ShareReduce, Opt-Proactivize and Opt-ShareDist. Other than the public inputs, the information obtained by the adversary \mathcal{A} is:

Opt-ShareReduce:

- For all corrupt \mathcal{U}_j in the previous handoff, reduced share $B(x, j)$.
- For all corrupt nodes \mathcal{C}_i in the old committee, $\left\{ B(i, j), W_{B(i, j)} \right\}_{j \in [2t+1]}$ (full share $B(i, y)$).
- For all corrupt \mathcal{U}'_j in the new committee selected to participate in the handoff, $\left\{ B(i, j), W_{B(i, j)} \right\}_{i \in [2t+1]}$ (reduced share $B(x, j)$).

Opt-Proactivize:

- For all corrupt nodes \mathcal{U}'_j, s_j and $Q(x, j) = R_j(x)$.
- For all corrupt nodes \mathcal{C}'_i in the new committee, H_j and $\left\{ g^{s_j}, C_{Z_j}, W_{Z_j(0)}, C_{B'(x, j)} \right\}$.

Opt-ShareDist:

- For all corrupt C'_i in the new committee, $\left\{ B'(i, j), W'_{B(i,j)} \right\}_{j \in [2t+1]}$.

The information above assumes the secrecy of our bivariate 0-sharing protocol, which we explained in the main body. In addition, note that the public information posted on chain are all commitments of the polynomials. By the hiding property of the commitment scheme based on the discrete log assumption, the PPT \mathcal{A} learns no extra information from these commitments. To prove secrecy, we have the following lemmas.

Lemma C.1. *If \mathcal{A} corrupts no more than t nodes in the old committee, and no more than t nodes in \mathcal{U}' , the information received by \mathcal{A} in Opt-ShareReduce is random and independent of the secret s .*

Proof. This is implied by the degree of the bivariate polynomial $B(x, y)$. In the worst case when all t corrupted nodes are in \mathcal{U} and \mathcal{U}' , \mathcal{A} learns $2t$ reduced shares $B(x, j)$ and t full shares $B(i, y)$. For a $\langle t, 2t \rangle$ -bivariate polynomial, any t shares of $B(i, y)$ and $2t$ shares of $B(x, j)$ are random and independent of $s = B(0, 0)$.

Moreover, based on the discrete-log assumption, the proofs $W_{B(i,j)}$ are computationally zero-knowledge by the KZG scheme, and the PPT adversary cannot learn additional information from them. \square

Lemma C.2. *Given a bivariate 0-sharing scheme with secrecy and integrity, if at least one node is honest in Opt-Proactivize, $Q(x, y)$ is randomly generated.*

Proof. Any $2t + 1$ degree t univariate polynomials $Q(x, j)$ uniquely define a $\langle t, 2t \rangle$ -bivariate polynomial. Therefore, as long as one node is honest and generates a random degree t polynomial, $Q(x, y)$ is randomly generated to mask $B(x, y)$.

Similar to the proof above, the hashes and commitments do not leak additional information to a PPT adversary \mathcal{A} .

□

Lemma C.3. *If \mathcal{A} corrupts no more than t nodes in the new committee \mathcal{C}' , the information received by \mathcal{A} in Opt-ShareDist is random and independent of the secret s .*

Proof. By Lemma C.1, $Q(x, y)$ is randomly generated, thus $B'(x, y) = B(x, y) + Q(x, y)$ is independent of $B(x, y)$. Regardless of the number of nodes corrupted by \mathcal{A} in \mathcal{U}' , \mathcal{A} receives no more than t out of n' shares of $B'(i, y)$ in Opt-ShareDist. As the degree of $B'(x, y)$ is $\langle t, 2t \rangle$ and is independent of $B(x, y)$, these shares are random and independent of s . Again, the proofs in the second part do not leak additional information. □

By Lemma C.1, C.2 and C.3, \mathcal{A} does not learn any information about s in consecutive epochs. The secrecy of the scheme follows by induction.

Integrity. For integrity, we have the following lemmas.

Lemma C.4. *After Opt-ShareReduce, at least $t + 1$ honest nodes \mathcal{U}'_j can successfully reconstruct $B(x, j)$.*

Proof. As the number of nodes in the old committee $n \geq 2t + 1$, each node \mathcal{U}'_j receives at least $t + 1$ correct shares of $B(i, j)$. As the degree on the first variable of $B(x, y)$ is t , \mathcal{U}'_j can reconstruct $B(x, j)$ successfully. Finally, as the number of nodes in \mathcal{U}' is $2t + 1$, there are at least $t + 1$ honest nodes. □

Lemma C.5. *Assuming the correctness of the bivariate 0-sharing scheme, after Opt-Proactivize, either honest nodes \mathcal{U}'_j hold the correct shares of $B'(x, j)$ such that $B'(0, 0) = B(0, 0) = s$ and their commitments $C_{B'(x, j)}$ are on-chain, or at least $t + 1$ honest nodes in \mathcal{C}' output fail.*

Proof. By line 15 in Figure 4.8, $\{g^{s_j}, C_{Z_j}, W_{Z_j(0)}, C_{B'(x, j)}\}$ is consistent with the hash H_j posted on chain by \mathcal{U}'_j . If C_{Z_j} is not a univariate polynomial with constant term 0, by line 16, VerifyEval outputs false and \mathcal{C}'_i outputs fail by the soundness of KZG. Otherwise, by the second check of line 16, $C_{B'(x, j)}$ is the commitment of a polynomial $B'(x, j)$ with constant term $B(x, j) + s_j$. Finally, by the check of line 17, by the discrete-log assumption, $\sum_{j=1}^{2t+1} s_j \lambda_j^{2t} = 0$. Therefore, $B'(0, 0) = B(0, 0)$ because of the property of Lagrange coefficients. \square

By Lemma C.4 and C.5, if Opt-ShareReduce and Opt-Proactivize do not fail, all nodes \mathcal{U}'_j hold the correct shares of $B'(x, j)$ such that $B'(0, 0) = B(0, 0) = s$ and their commitments $C_{B'(x, j)}$ are on the chain. In Opt-ShareDist, each node \mathcal{C}'_i receives $2t + 1$ shares of $B'(i, j)$ from all \mathcal{U}'_j s. By the soundness of the KZG scheme, if any of these shares is corrupt, VerifyEval rejects, and honest nodes in \mathcal{C}' output fail. Otherwise, with $2t + 1$ correct shares of $B'(i, j)$, \mathcal{C}'_i can successfully reconstruct $B'(i, y)$, which completes the proof of integrity.

C.2 CHURP Pessimistic paths

In this section, we present protocols for the two pessimistic paths of CHURP: Exp-CHURP-A and Exp-CHURP-B.

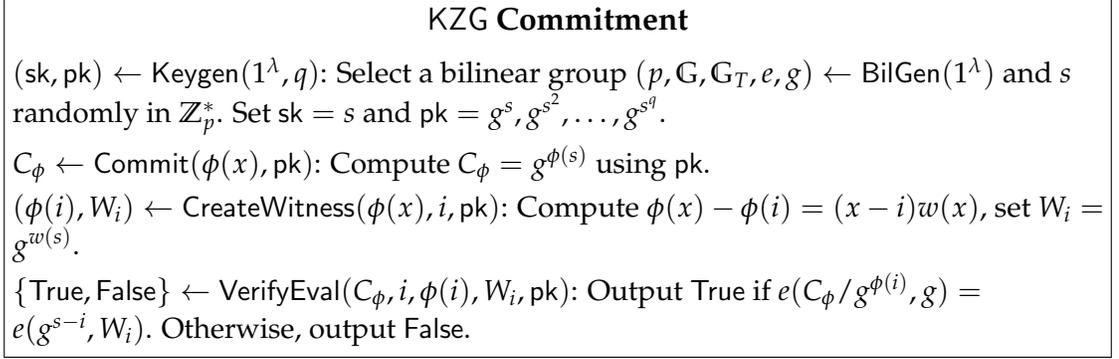


Figure C.1: Protocols of KZG commitment scheme.

C.2.1 Exp-CHURP-A

This path is invoked when a failure occurs in Opt-CHURP. As mentioned before, the pessimistic paths use on-chain communication only. The first phase is the same as Opt-ShareReduce, and is not re-executed if Opt-ShareReduce ends successfully.

In Exp-Proactivize, we use a different zero-sharing protocol, allowing honest parties to avoid re-execution of the protocol in case of corruption — they can simply discard the shares generated by the adversarial nodes. Messages are encrypted under the receiver’s public key and posted on-chain, so that a verifiable accusation can be performed in case of a corruption.

If any adversary in \mathcal{U}' is expelled in this phase, we ask members in the old committee to publish the shares and witnesses sent to the adversarial nodes during Opt-ShareReduce on-chain. Thus, all honest parties have access to reduced shares that belong to adversarial nodes, which allows them to reconstruct the full shares in the next phase.

In Exp-ShareDist, to allow identification of malicious nodes, members post all messages on-chain in contrast to the optimistic path. Exp-Proactivize and Exp-

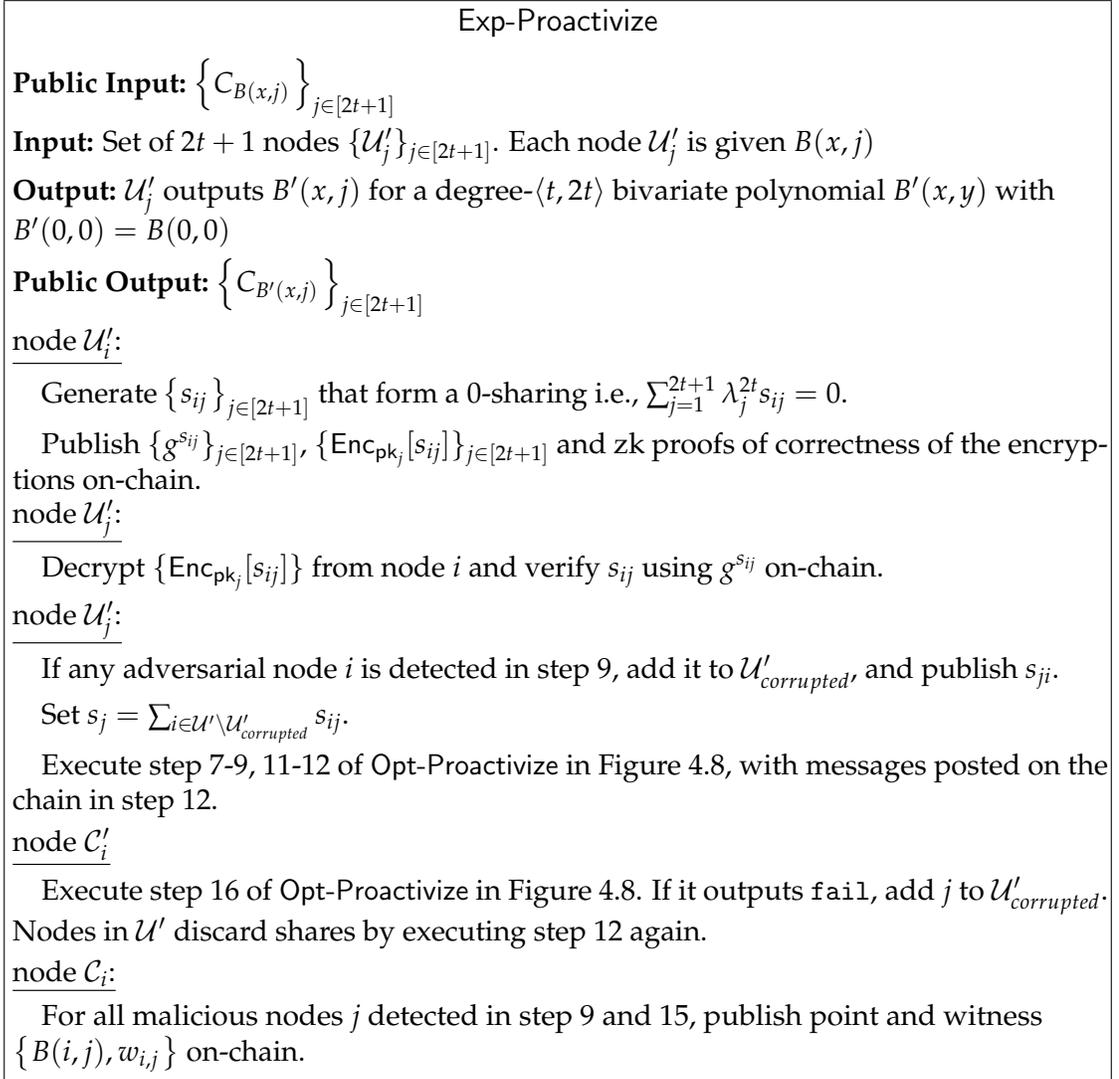


Figure C.2: Exp-Proactivize protocol.

ShareDist are presented in Figure C.2 and C.3. The overall on-chain complexity of Exp-CHURP-A is $O(n^2)$.

C.2.2 State Verification Details

Failure. There are two possible reasons that may cause StateVerif to fail: either the commitments are computed incorrectly by adversarial nodes, or the assumptions

in the KZG scheme fails. We further perform the following test to determine the reason.

We make use of the on-chain KZG commitments (published in CHURP) to verify the commitments $Z_i = g^{s_i}$ and $Z_i^{rnd} = g^{s'_i}$. Each node i posts exponents of their state $\{g^{B'(i,j)}\} \forall j \in [2t + 1]$, and their witness $w'_{j,i}$ to the KZG polynomial commitments $C_{B'(x,j)}$ on-chain (each node already has these witnesses at the

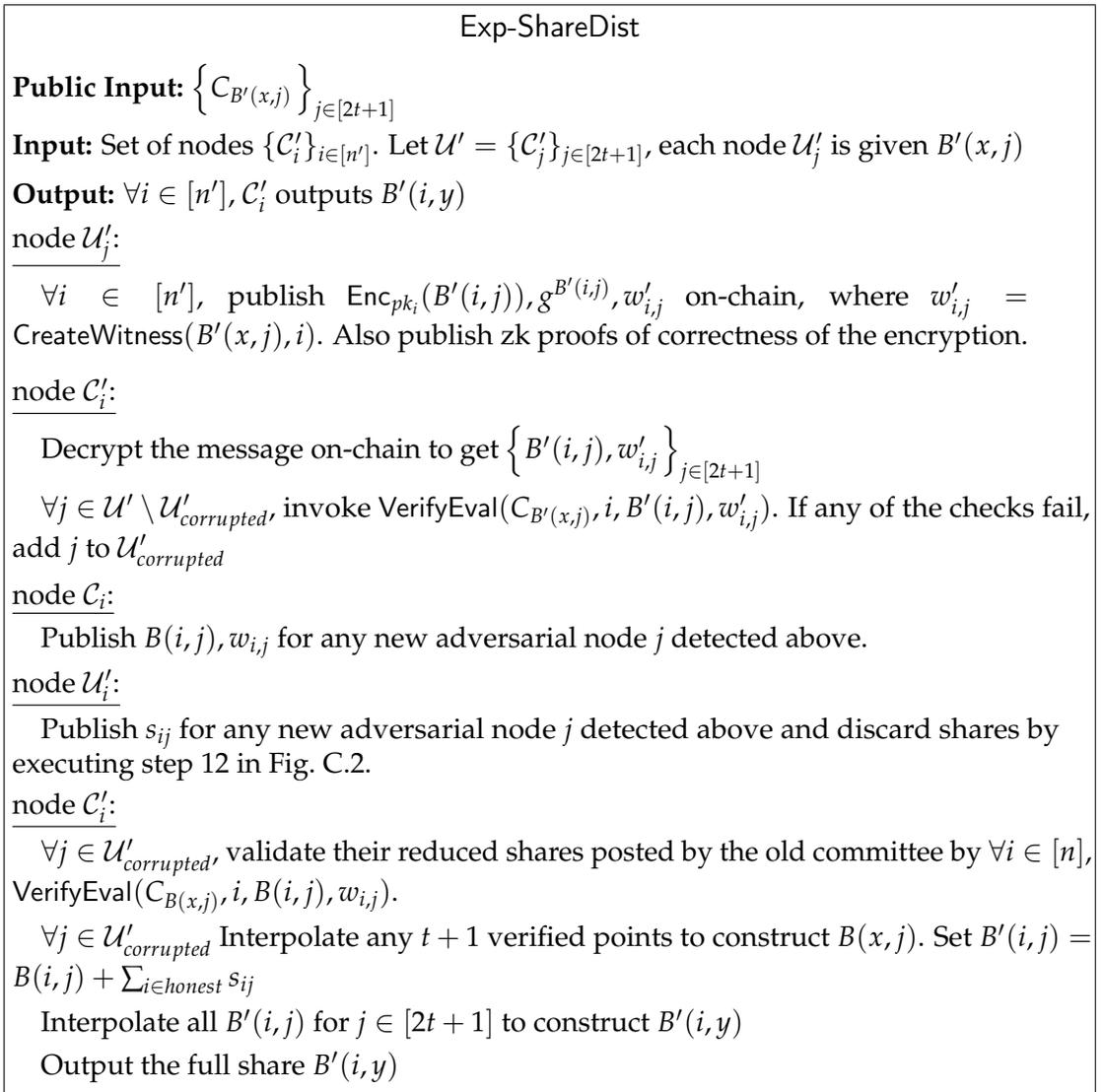


Figure C.3: Exp-ShareDist protocol.

end of Opt-CHURP or Exp-CHURP-A). Then all members verify the message published by node i : $\text{VerifyEvalExp}(C_{B'(x,i)}, i, g^{B'(i,j)}, W_{j,i})$ for $j \in [2t + 1]$. (We make use of the following additional functionality in KZG scheme that allows us to verify the exponent of the evaluation without any changes to the scheme: $\{\text{True}, \text{False}\} \leftarrow \text{VerifyEvalExp}(C_\phi, i, g^{\phi(i)}, W_i)$.)

If the checks above pass, all members validate Z_i, Z_i^{rnd} as: $Z_i = \prod_{j=1}^{2t+1} (g^{B'(i,j)})^{\lambda_j^{2t}}, Z_i^{rnd} = \prod_{j=1}^{2t+1} (g^{B'(i,j)})^{r_j \lambda_j^{2t}}$.

If any of the checks above fail, it means the commitments are not correctly computed. The members can perform a verifiable accusations since all information is on-chain, and then switch to pessimistic path Exp-CHURP-A. Otherwise, it implies a failure of the assumptions in the KZG scheme. In this case, we switch to a different pessimistic path Exp-CHURP-B. In this test, each node publishes $O(n)$ data on-chain, incurring $O(n^2)$ on-chain cost overall.

C.2.3 Exp-CHURP-B

This pessimistic path is taken only after a detection of breach in the underlying assumptions of the KZG scheme.

In this path, we use relatively expensive polynomial commitments (Pedersen commitments) instead of KZG and supports a lower threshold on the number of adversarial nodes $n > 3t$. In the share reduction phase, as $n > 3t$, we rely on the error correcting mechanisms of Reed-Solomon codes to construct reduced shares, instead of the verification of KZG scheme. In the proactivization phase and full share distribution phase, we replace the KZG commitments and verification with

the Pedersen commitments (step 13 in Figure C.2 and step 5,8,12 in Figure C.3). Exp-CHURP-B incurs $O(n^2)$ on-chain cost, assuming $n > 3t$. Due to the space limit, we omit the full protocol of Exp-CHURP-B.

C.3 The Static Setting: Improved PSS

We also consider a different and narrower setting, one with a *static* committee i.e., the old and new committees are identical. The adversarial model is also weaker i.e., corruptions during the handoff phase are counted towards the threshold in both the adjacent epochs. The handoff in such a setting is simply an *update* since the committee is static. Hence, the update protocol consists of a *recovery* phase, enabling recovery of lost shares and a *refresh* phase, updating shares of all nodes.

In this section, we look at different techniques seen in literature for the static setting. Herzberg et al. [145] introduce this setting and present a protocol, Herzberg’s PSS. A second technique seen in the literature makes use of bivariate polynomials. We then present an improved PSS protocol which achieves better overall performance than any known scheme.

Herzberg’s PSS: This protocol incurs $O(n^2)$ off-chain communication complexity for refresh and an expensive $O(n^2)$ per node recovery (See [145]).

Bivariate Polynomials: One way to avoid the expensive recovery cost is to perform secret sharing with a bivariate polynomial. This allows for efficient recovery, i.e., $O(n)$ off-chain communication complexity. As discussed previously in Section 4.4, existing techniques for refresh are expensive costing $O(n^3)$.

Improved PSS: Much like the dynamic setting, we build an improved PSS protocol

| | | Univariate [145] | Bivariate | Improved PSS |
|-----------|----------|------------------|-----------|--------------|
| Off-chain | Recovery | $O(n^2)$ | $O(n)$ | $O(n)$ |
| | Refresh | $O(n^2)$ | $O(n^3)$ | $O(n^2)$ |
| State | | $O(1)$ | $O(n)$ | $O(n)$ |

Table C.1: Comparison of protocols in the static setting with a honest-but-curious adversary. The original protocol of Herzberg et al. is presented in the univariate column. Recovery costs are per node. Note that recovery costs of our protocol are amortized over the total number of nodes being replaced.

using the efficient bivariate 0-sharing technique. This technique brings down the total communication complexity to just $O(n^2)$ off-chain. A comparison of communication costs incurred by different PSS schemes is in Table C.1.

Let \mathcal{C} denote the committee, $\mathcal{C} = \mathcal{C}^{(e-1)} = \mathcal{C}^{(e)}$, comprising n nodes $\{\mathcal{C}_i\}_{i=1}^n$. The secret is shared using an asymmetric bivariate polynomial $B(x, y)$, $s = B(0, 0)$. Unlike before, the degree of bivariate polynomial is only $\langle t, t \rangle$ as we have a weaker adversary.

Recall that node's share is a single polynomial $B(i, y)$. In Fig. C.4, we present the improved PSS assuming a honest-but-curious adversary. Throughout the protocol, each node sends out atmost $O(n)$ points. Thus, our improved PSS scheme completes in $O(n^2)$ off-chain cost.

Active adversaries: In face of adversarial behaviour, multiple reruns of the protocol might be needed. This is crucial since all the $t + 1$ received points need to be correct in order to compute the new share. Adversaries are detectable with the use of KZG commitments similar to the dynamic setting. We replace the detected adversarial nodes with uncorrupted nodes from \mathcal{C} (guaranteed to find such a node, $|\mathcal{C}| \geq 2t + 1$). We stress that this protocol incurs $O(n^2)$ off-chain cost even after adapting to handle active adversaries. This is achieved due to the

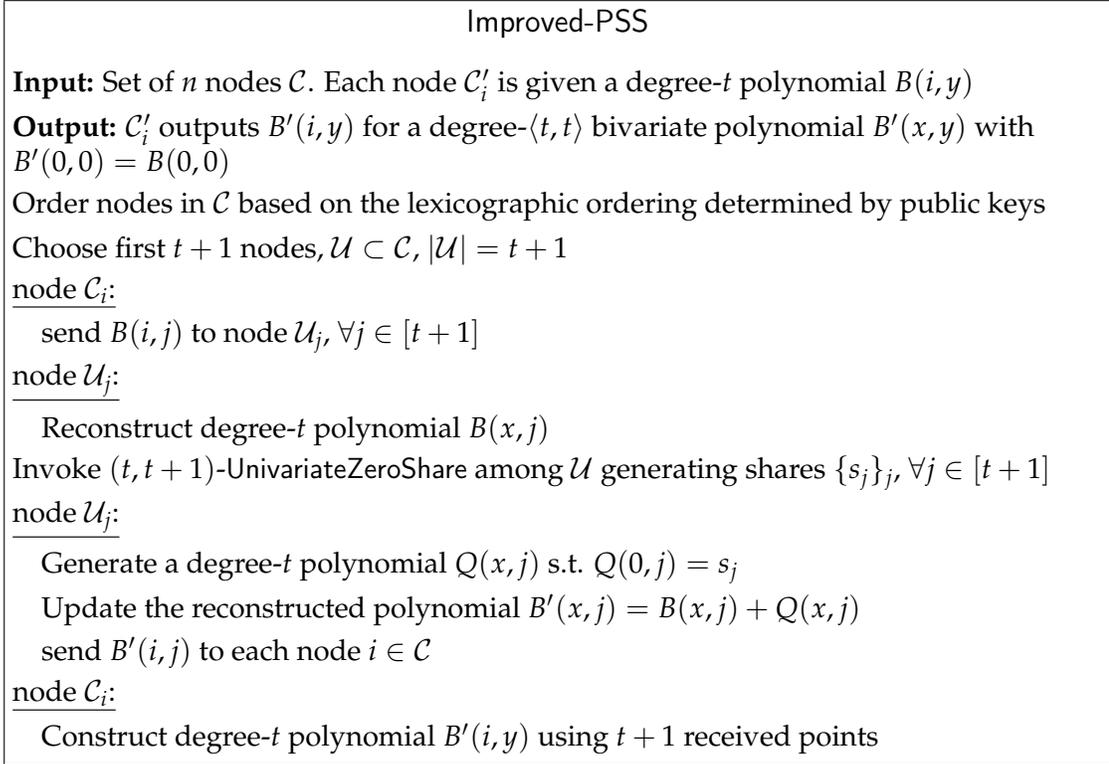


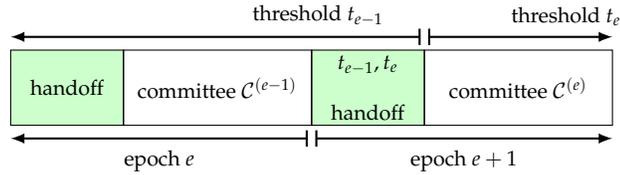
Figure C.4: Improved PSS for static setting, honest-but-curious adversary.

following key property: Honest nodes never rerun any phase of the protocol. This is possible by making a slight modification to the univariate 0-sharing (step 9): invoke (t, n) -UnivariateZeroShare among all nodes in \mathcal{C} instead of executing it in a subset of nodes only. Observe that the set of univariate polynomials held by any $t + 1$ -sized subset in \mathcal{C} defines a 0-hole bivariate polynomial. Thus, reruns are executed only by the new uncorrupt nodes that replace the detected faulty nodes.

C.4 Changing the threshold

C.4.1 Increasing the threshold: $t_e > t_{e-1}$

Note that a change of the threshold reflects that of the adversary's power, i.e., the number of nodes it can corrupt in the committee $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$, respectively. Therefore extra care is needed if we were to increase the power of the adversary (i.e. $t_e > t_{e-1}$). Similar to [226], increasing the threshold takes two steps: first, a handoff is executed between $\mathcal{C}^{(e-1)}$ and $\mathcal{C}^{(e)}$ assuming the threshold doesn't change; then we increase the threshold to t_e after the handoff. As illustrated below, the new threshold takes effect *after* the handoff.



Specifically, to increase the threshold, (t_{e-1}, t_e) -handoff runs the proactivization phase with parameters $t = t_e$. That is, during the proactivization protocol, a bivariate polynomial $Q(x, y)$ of degree $(t_e, 2t_e)$ is generated such that $Q(0, 0) = 0$. Each node i holds a t_e -degree polynomial $Q(x, i)$ and commitments to $\{Q(x, i)\}_i$ are publicly available. The rest of the proactivization follows without modification, besides now each node i holds two polynomials with different degrees: $B'(x, i)$ that is t_{e-1} -degree while $Q(x, i)$ is t_e -degree. Thus the proactivized global polynomial $B'(x, y)$ is of degree $(t_e, 2t_e)$, concluding the threshold upgrade.

We also need to extend KZG to support dynamic thresholds, i.e., given a commitment C_ϕ , it can be publicly verified that ϕ is at most d -degree. Essentially,

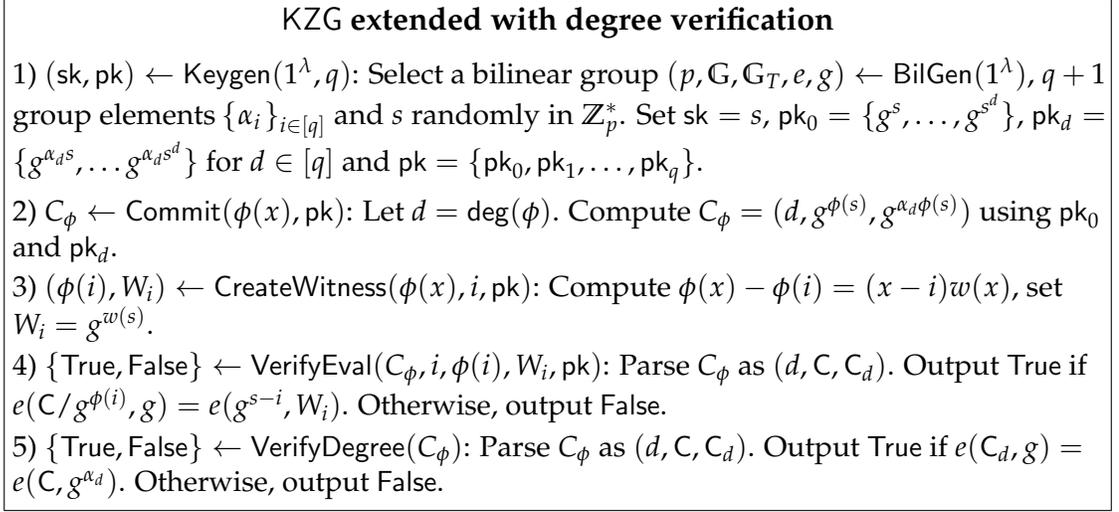


Figure C.5: KZG [162] extended with degree verification.

the setup phase of the KZG fixes the highest degree (say, D) of polynomials it can work with. In the setting of a static threshold t , we set $D = t$ and a KZG commitment can guarantee that hidden polynomials are of degree $\leq t$, which is critical to the correctness of shares. To support dynamic thresholds up to t_{\max} , we extend KZG as specified in Fig. C.5 and run the trusted setup with $D = t_{\max}$. Our extension relies on the q -PKE [138] assumption.

C.4.2 Decreasing the threshold

The intuition of decreasing the threshold is to create $2 \times (t_{e-1} - t_e)$ *virtual nodes*, denoted as \mathcal{V} , and execute the handoff protocol between $\mathcal{C} = \mathcal{C}^{(e-1)}$ and $\mathcal{C}' = \mathcal{C}^{(e)} \cup \mathcal{V}$, assuming the threshold remains t_{e-1} . A virtual node participates in the protocol as if an honest player, but exposes its state publicly. At the end of the handoff protocol, nodes in \mathcal{C}' incorporate \mathcal{V} 's state and restore the invariants. The handoff protocol is outlined as follows.

Decreasing the threshold

- 1) Choose a subset $\mathcal{U} \subseteq \mathcal{C}'$ of $2t_e + 1$ nodes. For notational simplicity, suppose $\mathcal{U} = \{1, \dots, 2t_e + 1\}$ and $\mathcal{V} = \{2t_e + 2, \dots, 2t_{e-1} + 1\}$. Each node $i \in \mathcal{U}$ recovers a reduced share $RS_i^{(e-1)}(x) = B(x, i)$. In addition, \mathcal{C} publishes reduced shares for virtual nodes: $RS_j^{(e-1)}(x) = B(x, j)$ for $j \in \mathcal{V}$.
- 2) \mathcal{U} executes the proactivization phase and collectively generate a $(t_e, 2t_e)$ -degree bivariate zero-hole polynomial $Q(x, y)$. At the end of this phase, each node $i \in \mathcal{U}$ has $Q(x, i)$.
- 3) Let $V = \sum_{j \in \mathcal{V}} \lambda_j^{2t_{e-1}} RS_j^{(e-1)}(0)$. Each node $i \in \mathcal{U}$ incorporates virtual nodes' state and updates its state as $RS_i^{(e)}(x) = \frac{\lambda_i^{2t_{e-1}}}{\lambda_i^{2t_e}} \left(RS_i^{(e-1)}(x) + \frac{V}{\lambda_i^{2t_{e-1}(2t_e+1)}} \right) + Q(x, i)$ where $\lambda^{2t_{e-1}}$ and λ^{2t_e} are Lagrange coefficients for corresponding thresholds. One can verify that $RS_i^{(e)}(x)$ are $2t_e$ -sharing of the secret, i.e., $B(0, 0)$ can be calculated from any $2t_e + 1$ of $RS_i^{(e)}(x)$.
- 4) Each node $i \in \mathcal{U}$ sends to every node $j \in \mathcal{C}'$ a point $RS_i^{(e)}(j)$. The full share of each node $j \in \mathcal{C}'$ consists of $2t_e + 1$ points $\{RS_i^{(e)}(j) = B'(i, j)\}_{i \in \mathcal{U}}$, from which j can compute $FS_j(y) = B'(j, y)$.

The updated reduced shares $RS_i^{(e)}(x)$ can be verified using the published value V , and the commitment to $RS_i^{(e-1)}(x)$ and $Q(x, i)$. At the end, each node i has $2t_e + 1$ points on $B'(i, y)$. It remains to show that $\{FS_j(y) = B'(j, y)\}_j$ form a t_e -sharing of $B^{(e)}(0, 0)$, which can be checked by $\sum_{i=1}^{t_e+1} \lambda_i^{t_e} FS_i(0) = \sum_{j=1}^{2t_{e-1}+1} \lambda_j^{2t_{e-1}} RS_j^{(e-1)}(0) = B(0, 0)$.

Several optimizations are possible. For example, one can reduce the degree of $RS_i^{(e)}(x)$ to t_e (as opposed to t_{e-1} currently) by building new polynomials and proving equivalence to $RS_i^{(e-1)}(x)$. We leave further optimization for future work.

APPENDIX D

REM

D.1 Tolerating Compromised SGX Nodes: Details

D.1.1 Mining Rate Estimation

We start by discussing how to statistically infer the power of a CPU from its blocks in the blockchain. Reading the difficulty of each block in the main chain and the rate of blocks from a specific CPU, we can estimate a lower bound of that CPU's power – it follows directly from the rate of its blocks. It is a lower bound since the CPU might not be working continuously, and the estimate's accuracy increases with the number of available blocks.

Recall C_{m_i} is the blocks mined by miner m_i so far. C_{m_i} may contain multiple blocks, perhaps with varying difficulties. Without loss of generality, we write the difficulty as a function of time, $d(t)$. The difficulty is the probability for a single instruction to yield a win. Denote the power of the miner, i.e., its mining rate, by rate_i . Therefore in a given time interval of length T , the number of blocks mined by a specific CPU obeys Poisson distribution (since CPU rates are high and the win probability is small, it's appropriate to approximate a Binomial distribution by a Poisson distribution,) and with rate $\text{rate}_i T d(t)$. Further, under independence assumption, the mining process of a specific CPU is specified by a Poisson process with rate $\lambda_i(t) = \text{rate}_i d(t)$, the product of the probability and the miner's rate rate_i .

There are many methods to estimate the mean of a Poisson distribution.

Knowing rates for all miners, the rate of the strongest CPU ($\text{rate}_{\text{best}}$) can be estimated. The challenge here is to limit the influence of adversarial nodes. To this end, instead of finding the strongest CPU directly, we approximate $\text{rate}_{\text{best}}$ based on rate_ρ (e.g. $f_{90\%}$), namely the ρ -percentile fastest miner.

Bootstrapping. During the launch of a cryptocurrency, it could be challenging to estimate the mining power of the population accurately, potentially leading to poisoning attacks by an adversary. At this early stage, it makes sense to hardwire a system estimate of the maximum mining power of honest miners into the system and set conditions (e.g., a particular mining rate or target date) to estimate $\text{rate}_{\text{best}}$ as we propose above. If the cryptocurrency launches with a large number of miners, an even simpler approach is possible before switching to $\text{rate}_{\text{best}}$ estimation: We can cap the total number of blocks that any one node can mine, a policy we illustrate below. (See P_{simple} .)

D.1.2 Security game definition

We model REM as an interaction among three entities: a blockchain consensus algorithm, an adversary, and a set of honest miners. Their behavior together defines a *security game*, which we define formally below. We characterize the three entities respectively as (ideal) programs $\text{prog}_{\text{chain}}$, $\text{prog}_{\mathcal{A}}$, and prog_m , which we now define.

Blockchain consensus algorithm ($\text{prog}_{\text{chain}}$). A consensus algorithm determines which valid blocks are added to a blockchain C . We assume that underlying consensus and fork resolution are instantaneous; loosening this assumption does

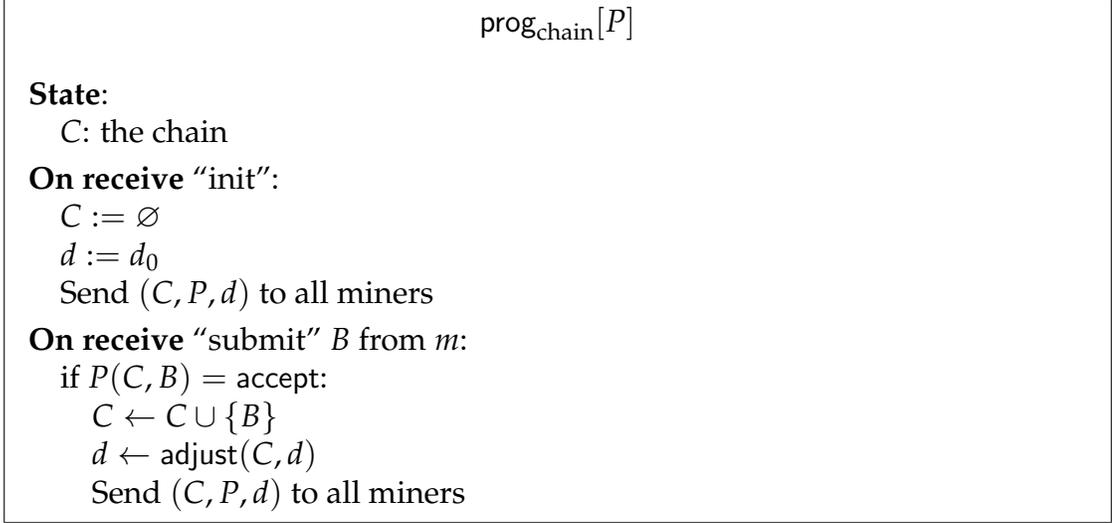


Figure D.1: The program for a blockchain. We omit details here on how difficulty d is set, i.e., how d_0 and adjust are chosen.

not materially affect our analyses. We also assume that block timestamping is accurate. Timestamps can technically be forged at block generation, but in practice miners reject blocks with large skews [54], limiting the impact of timestamp forgery.

Informally, $\text{prog}_{\text{chain}}$ maintains and broadcasts and authoritative blockchain C . In addition to verifying that block contents are correct, $\text{prog}_{\text{chain}}$ appends to C only blocks that are valid under a policy P . We model the blockchain consensus algorithm as the (ideal) stateful program specified in Fig. D.1.

Adversary \mathcal{A} ($\text{prog}_{\mathcal{A}}$). In our model, an adversary \mathcal{A} executes a *strategy* $\Sigma_{\mathcal{A}}$ that coordinates the k miners $M_{\mathcal{A}}$ under her control to generate blocks. Specifically:

Definition D.1. (*Adversarial Strategy*). An adversarial strategy is a probabilistic algorithm $\Sigma_{\mathcal{A}}$ that takes in a set of identities, the current blockchain and the policy, and outputs a time-stamp and identity for block submission. Specifically, $(M_{\mathcal{A}}, C, t, P) \rightarrow (\hat{t}, \hat{m}) \in \mathbb{R}^+ \times M_{\mathcal{A}}$.

$\text{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}]$

On receive (C, P, d) from $\text{prog}_{\text{chain}}$
 $\hat{t}, \hat{m} \leftarrow \Sigma_{\mathcal{A}}(M_{\mathcal{A}}, C, P, d)$
 if \hat{t} is not \perp :
 wait until \hat{t}
 send “submit” (\hat{t}, \hat{m}, d) to $\text{prog}_{\text{chain}}$

Figure D.2: The program for an adversary \mathcal{A} that controls k nodes $M_{\mathcal{A}} = \{m_{\mathcal{A}1}, \dots, m_{\mathcal{A}k}\}$.

$\text{prog}_m[\Sigma_h]$

On receive (C, P, d) from $\text{prog}_{\text{chain}}$
 $\hat{t} \leftarrow \Sigma_h(C, d)$
 Send “submit” (\hat{t}, m, d) to $\text{prog}_{\text{chain}}$

Figure D.3: The program for an honest miner. Σ_h is the protocol defined by $\text{prog}_{\text{chain}}$ (e.g. PoET or PoUW).

In principle, $\Sigma_{\mathcal{A}}$ can have dependencies among individual node behaviors. In our setting, this would not benefit \mathcal{A} , however. As we don’t know $M_{\mathcal{A}}$ *a priori*, though, the only policies we consider operate on individual miner block-generation history.

As a wrapper expressing implementation by \mathcal{A} of $\Sigma_{\mathcal{A}}$, we model \mathcal{A} as a program $\text{prog}_{\mathcal{A}}$, specified in Fig. D.2.

Honest miners (prog_m). Every honest miner $m \in M - M_{\mathcal{A}}$ follows an identical strategy, a probabilistic algorithm denoted Σ_h . In REM, Σ_h may be modeled as a simple algorithm that samples from a probability distribution on block mining times determined by $\text{rate}(m)$ (specifically in our setting, an exponential distribution with rate $\text{rate}(m)$). We express implementation by honest miner m of Σ_h as a program $\text{prog}_m[\Sigma_h]$ (Fig. D.3).

```

 $P_{\text{simple}}(C, B):$ 
  parse  $B \rightarrow (\tau, m, d)$ 
  if  $|C_m| > 0$ :
    output reject
  else
    output accept

```

Figure D.4: A simple policy that allows one block per CPU over its lifetime.

To understand the security of REM, we consider a *security game* that defines how an adversary \mathcal{A} interacts with honest miners, a blockchain consensus protocol, and a policy given the above three ideal programs. Formally:

Definition D.2. (*Security Game*) For a given triple of ideal programs $(\text{prog}_{\text{chain}}[P], \text{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}], \text{prog}_m[\Sigma_h])$ and policy P , a security game $S(P)$ is a tuple $S(P) = ((M, M_{\mathcal{A}}, \text{rate}(\cdot)); (\Sigma_{\mathcal{A}}, \Sigma_h))$.

We define the *execution* of $S(P)$ as an interactive execution of programs $(\text{prog}_{\text{chain}}[P], \text{prog}_{\mathcal{A}}[\Sigma_{\mathcal{A}}], \text{prog}_m[\Sigma_h])$ using the parameters of $S(P)$. As P , $\Sigma_{\mathcal{A}}$ and Σ_h are randomized algorithms, such execution is itself probabilistic. Thus we may view the blockchain resulting from execution of S for interval of time τ as a random variable $C_S(\tau)$.

A *non-degenerate* security game S is one in which there exists at least one honest miner m with $\text{rate}(m) > 0$.

D.1.3 Warmup policy

As a warmup, we give a simple example of a potential block-acceptance policy. This policy just allows one block throughout the life of a CPU, as shown in Figure D.4.

Clearly, an adversary cannot do better than mining one block. Denote this simple strategy Σ_{simple} . For any non-degenerate security game S , therefore, the advantage $\text{Adv}_{\mathcal{A}}^{S(P_{\text{simple}})}(\tau) = 1$ as $\tau \rightarrow \infty$. This policy is optimal in that an adversary cannot do better than an honest miner unconditionally. However the asymptotic waste of this policy is 100%.

Another disadvantage of this policy is that it discourages miners from participating. Arguably, a miner would stay if the revenue from mining is high enough to cover the cost of replacing a CPU. But though a CPU is still valuable in other contexts even if it is blacklisted forever in *this* particular system, repurposing it incurs operational cost. Therefore chances are this policy would cause a loss of mining power, especially when the initial miner population is small, rendering the system more vulnerable to attacks.

D.1.4 Adversarial advantage

A block-acceptance policy depends only on the number of blocks by the adversary since its first one. Therefore an adversary's best strategy is simply to publish its blocks as soon as they won't be rejected. Denote this strategy as Σ_{stat} .

Clearly, an adversary will submit $F^{-1}(1 - \alpha, td \cdot \text{rate}_{\text{best}})$ blocks within $[0, t]$. On the other hand, the strongest honest CPU with rate $\text{rate}_{\text{best}}$ mines $td \cdot \text{rate}_{\text{best}}$ blocks in expectation. Recall that according to our Markov chain analysis, P_{stat} incurs false rejections for honest miners with probability $w_h(\alpha)$, which further reduces the payoff for honest miners. For a (non-degenerate) security game S , in which \mathcal{A} uses strategy Σ_{stat} , the advantage is therefore:

$$\text{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^\alpha)} = \lim_{t \rightarrow \infty} \frac{F^{-1}(1 - \alpha, td \cdot \text{rate}_{\text{best}})}{(1 - w_h(\alpha)) td \cdot \text{rate}_{\text{best}}} \quad (\text{D.1})$$

Theorem D.1. *In a (non-degenerate) security game S where \mathcal{A} uses strategy Σ_{stat} ,*

$$\text{Adv}_{\mathcal{A}}^{S(P_{\text{stat}}^\alpha)} = \frac{1}{1 - \text{Waste}(P_{\text{stat}}^\alpha)}.$$

Proof. Let $\lambda = td \cdot \text{rate}_{\text{best}}$. It is known that as λ for a Poisson distribution goes to infinity, it converges in the limit to a normal distribution with mean and variance λ . Therefore,

$$\lim_{\lambda \rightarrow \infty} \frac{F^{-1}(1 - \alpha, \lambda)}{(1 - w_h(\alpha)) \lambda} = \lim_{\lambda \rightarrow \infty} \frac{\lambda + \sqrt{\lambda} z_p}{(1 - w_h(\alpha)) \lambda} = \frac{1}{1 - w_h(\alpha)}.$$

□

Early in a blockchain's evolution, the potential advantage of an adversary is relatively high. The confidence interval is wide at this point, allowing the adversary to perform frequent generation without triggering detection. As the adversary publishes more blocks, the confidence interval tightens, forcing the adversary to reduce her mining rate. This is illustrated by our numerical simulation in Section 5.4.3.

BIBLIOGRAPHY

- [1] A Lossless, High Performance Implementation of the Zlib (RFC 1950) and Deflate (RFC 1951) Algorithm. <https://code.google.com/archive/p/miniz/>. Accessed: 2017-2-16.
- [2] Amazon EC2 instance pricing. <https://aws.amazon.com/ec2/instance-types://aws.amazon.com/ec2/pricing/on-demand/>. Accessed: 2016-10-29.
- [3] Amazon EC2 instance types. <https://aws.amazon.com/ec2/instance-types/>. Accessed: 2016-10-29.
- [4] Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation. Revision 2.0. Section 4.2.2. <https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf>. Accessed: 2017-2-21.
- [5] Augur. <http://www.augur.net/>.
- [6] Binary option. https://en.wikipedia.org/wiki/Binary_option.
- [7] Ethereum gas price tracker. <https://etherscan.io/gastracker>.
- [8] Ethereum gas station. <https://ethgasstation.info/>. Accessed: 11/13/2018.
- [9] An example of stale CPU dealers on Alibaba. <https://wefound.en.alibaba.com/>. Accessed: 2016-10-29.
- [10] Go language interface to gmp - gnu multiprecision library. <https://github.com/ncw/gmp>.
- [11] gRPC: A high performance, open-source universal RPC framework. <https://grpc.io>.
- [12] Intel(R) Software Guard Extensions for Linux OS. <https://github.com/01org/linux-sgx>.
- [13] Intel® Software Guard Extensions Enclave Writer's Guide. <https://software.intel.com/sites/default/files/managed/ae/48/>

Software-Guard-Extensions-Enclave-Writers-Guide.pdf. Accessed: 2017-2-16.

- [14] libsnark. <https://github.com/scipr-lab/libsnark>.
- [15] Open id connect.
- [16] Passmark software. <https://www.cpubenchmark.net/>. Accessed: 2016-10-29.
- [17] The PBC Go wrapper. <https://github.com/Nik-U/pbc>.
- [18] PriceFeed smart contract. <http://feed.ether.camp/>. Accessed Feb. 2016.
- [19] Sawtooth-core source code (validator). https://github.com/hyperledger/sawtooth-core/tree/0-7/validator/sawtooth_validator/consensus/poet1. Accessed: 2017-2-21.
- [20] Secure Online Age Verification • AgeChecker.Net. <https://agechecker.net>.
- [21] Single-file C implementation of the SHA-3 implementation with Init/Update/Finalize hashing (NIST FIPS 202). <https://github.com/brainhub/SHA3IUF>. Accessed: 2017-2-16.
- [22] Steam online gaming platform. <http://store.steampowered.com/>.
- [23] TLSNotary. <https://tlsnotary.org/>.
- [24] Uniswap: A fully decentralized protocol for automated liquidity provision on Ethereum. <https://uniswap.org>.
- [25] Art. 20, GDPR, Right to data portability, 2014. <https://gdpr-info.eu/art-20-gdpr/>.
- [26] uport. <https://www.uport.me/>, 2018.
- [27] Provable blockchain oracle. <http://provable.xyz>, 2019.
- [28] (Bristol Format) circuits of basic functions suitable for MPC, Aug 2019. <https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html>.

- [29] Oraclize: The provably honest oracle service. <http://www.oraclize.it>, Referenced Feb. 2016.
- [30] John Adler, Ryan Berryhill, Andreas G. Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A decentralized blockchain oracle. In *IEEE iThings/GreenCom/CPSCoM/SmartData*, 2018.
- [31] Yazin Akkawi. Bitcoin’s most pressing issue summarized in two letters: UX, Dec 2017.
- [32] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.
- [33] David P. Anderson. BOINC: A platform for volunteer computing. volume 18, pages 99–122, 2020.
- [34] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient Library for Cryptography. <https://github.com/relic-toolkit/relic>.
- [35] ARM. mbedTLS: An open source, portable, easy to use, readable and flexible SSL library. <https://github.com/ARMmbed/mbedtls>, 2019.
- [36] Brian Armstrong. Coinbase is not a wallet. *Coinbase blog*, Feb. 25, 2018.
- [37] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. Helix: a scalable and fair consensus algorithm. Technical report, Orbs Research, 2018.
- [38] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged Byzantine impostors. *Department of Computer Science, Yale University, New Haven, CT, Tech. Rep*, 2005.
- [39] American Bar Association. <https://www.americanbar.org/>.
- [40] Giuseppe Ateniese, Daniel H. Chou, Breno de Medeiros, and Gene Tsudik. Sanitizable signatures. In *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2005.
- [41] Microsoft Azure. Blockchain as a service. <https://web.archive.org/web/>

20161027013817/https://azure.microsoft.com/en-us/solutions/blockchain/, 2016.

- [42] Adam Back et al. Hashcash-a denial of service counter-measure. Technical report, 2002.
- [43] Michael Backes, Amit Datta, and Aniket Kate. Asynchronous computational VSS with reduced communication complexity. In *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2013.
- [44] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *PODC*, pages 201–209. ACM, 1989.
- [45] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2015.
- [46] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, pages 267–283. USENIX Association, 2014.
- [47] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In *ASIACRYPT*, pages 232–249. Springer, 2009.
- [48] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
- [49] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.
- [50] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2016.
- [51] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of

activity: Extending bitcoin's proof of work via proof of stake [extended abstract]y. *SIGMETRICS Perform. Evaluation Rev.*, 42(3):34–37, 2014.

- [52] Bhavya Bhandari. Supply Chain Management, Blockchains and Smart Contracts. SSRN Scholarly Paper ID 3204297, Social Science Research Network, Rochester, NY, June 2018.
- [53] Robert Biddle, Paul C. van Oorschot, Andrew S. Patrick, Jennifer Sobey, and Tara Whalen. Browser interfaces and extended validation SSL certificates: an empirical study. In *CCSW*, pages 19–30. ACM, 2009.
- [54] Bitcoin community. Bitcoin source. <https://github.com/bitcoin/bitcoin>. Retrieved Nov. 2016.
- [55] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS). RFC 4492, May 2006.
- [56] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. *IACR ePrint Archive*, 2015:1015, 2015.
- [57] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security*, pages 187–198. ACM, 2009.
- [58] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID from Bilinear Pairing. *IACR Cryptology ePrint Archive*, 2009:95, 2009.
- [59] Mark Brown and Russ Housley. Transport layer security (TLS) evidence extensions, 2007. <https://tools.ietf.org/html/draft-housley-evidence-extns-01>.
- [60] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schröder, and Florian Volk. Security of sanitizable signatures revisited. In *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009.
- [61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom

- with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008. USENIX Association, 2018.
- [62] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society, 2018.
- [63] Vitalik Buterin. Schellingcoin: A minimal-trust universal data feed. *Ethereum blog*.
- [64] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. <https://github.com/ethereum/wiki/wiki/White-Paper>, 2014.
- [65] Vitalik Buterin. Slasher: A punitive proof-of-stake algorithm. 2014.
- [66] Christian Cachin et al. Architecture of the Hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.
- [67] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strohli. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, pages 88–97. ACM, 2002.
- [68] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 1997.
- [69] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *ACM Conference on Computer and Communications Security*, pages 229–243. ACM, 2017.
- [70] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
- [71] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.

- [72] Ran Canetti and Tal Rabin. Universal composition with joint state. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2003.
- [73] Miles Carlsten, Harry A. Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In *ACM Conference on Computer and Communications Security*, pages 154–167. ACM, 2016.
- [74] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- [75] Mark Cavage and Manu Sporny. Signing HTTP Messages. Internet-Draft draft-cavage-http-signatures-11, Internet Engineering Task Force, April 2019. Work in Progress.
- [76] CFTC. A primer on smart contracts. https://www.cftc.gov/sites/default/files/2018-11/LabCFTC_PrimerSmartContracts112718.pdf, Nov 2018.
- [77] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [78] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.
- [79] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *EuroS&P*, pages 185–200. IEEE, 2019.
- [80] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.
- [81] Ethereum community. Devp2p. <https://github.com/ethereum/devp2p>.
- [82] Ethereum community. go-ethereum. <https://github.com/ethereum/go-ethereum>.

- [83] Ethereum community. The maximum data size in a transaction is 32 KB. https://github.com/ethereum/go-ethereum/blob/6a33954731658667056466bf7573ed1c397f4750/core/tx_pool.go#L570. Referenced on Nov 9 2018.
- [84] The Nxt Community. Nxt whitepaper, revision 4. https://web.archive.org/web/20160207083400/https://www.dropbox.com/s/cbuwrorf672c0yy/NxtWhitepaper_v122_rev4.pdf, 2014.
- [85] Nicholas Confessore. Cambridge analytica and Facebook: The scandal and the fallout so far. *The New York Times*, Apr 2018.
- [86] Microsoft Corp. Decentralized identity: Own and control your identity. <https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE2DjfY>, 2018.
- [87] Intel Corporation. Sawtooth lake – introduction. <https://web.archive.org/web/20161025232205/https://intelledger.github.io/introduction.html>, 2016.
- [88] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.
- [89] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.
- [90] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Eur. Trans. Telecommun.*, 8(5):481–490, 1997.
- [91] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 106–125. Springer, 2016.
- [92] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography*, volume 11598 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2019.

- [93] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.
- [94] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261. Springer, 2008.
- [95] Decentralized Identity Foundation. <https://identity.foundation/>, 2018.
- [96] Sebastiaan Deetman. Bitcoin could consume as much electricity as Denmark by 2020. *Motherboard*, March 2016.
- [97] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 235–254. IEEE Computer Society, 2016.
- [98] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *ACM Conference on Computer and Communications Security*, pages 1504–1517. ACM, 2015.
- [99] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures (extended abstract). In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 1991.
- [100] Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications. Technical report, 1997.
- [101] Yvo Desmedt and Kirill Morozov. Parity check based redistribution of secret shares. In *ISIT*, pages 959–963. IEEE, 2015.
- [102] Diar. Venture capital firms go deep and wide with blockchain investments. *Diar*, Oct 2018. <https://diar.co/volume-2-issue-39/#2>.
- [103] Digiconomist. Bitcoin Energy Consumption Index, May 2020. [Online; accessed 17. May 2020].

- [104] Shlomi Dolev, Juan A. Garay, Niv Gilboa, and Vladimir Kolesnikov. Brief announcement: swarming secrets. In *PODC*, pages 231–232. ACM, 2010.
- [105] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNacls: Portable native client executables. 2010.
- [106] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM Conference on Computer and Communications Security*, pages 1197–1210. ACM, 2015.
- [107] Thaddeus Dryja. Optimal mining strategies. SF Bitcoin-Devs presentation. <https://www.youtube.com/watch?v=QN2TPeQ9mnA>, 2014.
- [108] Tuyet Duong, Lei Fan, Thomas Veale, and Hong-Sheng Zhou. Securing Bitcoin-like backbone protocols against a malicious majority of computing power. Cryptology ePrint Archive, Report 2016/716, 2016. <http://eprint.iacr.org/2016/716>.
- [109] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.
- [110] Morris J Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Technical report, 2007. SP 800-38D.
- [111] John PR Dwyer and Patricia Hines. Beyond the byzz: Exploring distributed ledger technology use cases in capital markets and corporate banking. Technical report, Celent and MISYS, 2016.
- [112] Michael Egorov, MacLane Wilkison, and David Nuñez. Nucypher KMS: decentralized key management system. *arXiv preprint arXiv:1707.06140*, 2017.
- [113] Steve Ellis, Ari Juels, and Sergey Nazarov. Chainlink: A decentralized oracle network. <https://link.smartcontract.com/whitepaper>, 4 Sept. 2017.
- [114] Etherscan. Ethereum average gasprice chart. <https://etherscan.io/chart/gasprice>. [Online; accessed 2018].

- [115] Ittay Eyal. The miner’s dilemma. In *IEEE Symposium on Security and Privacy*, pages 89–103, 2015.
- [116] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59. USENIX Association, 2016.
- [117] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454. Springer, 2014.
- [118] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–437. IEEE Computer Society, 1987.
- [119] Peasech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.
- [120] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Optimal resilience proactive public-key cryptosystems. In *FOCS*, pages 384–393. IEEE Computer Society, 1997.
- [121] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Proactive RSA. In *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 1997.
- [122] Aviva Freudmann. From concept to reality: How blockchain will reshape the financial services industry. *The Economist*.
- [123] frontrun.me. Visualizing Ethereum gas auctions. <http://frontrun.me/>.
- [124] FTC. Price discrimination: Robinson-patman violations. <https://www.ftc.gov/tips-advice/competition-guidance/guide-antitrust-laws/price-discrimination-robinson-patman>, 2017.
- [125] Georg Fuchsbauer. Subversion-zero-knowledge snarks. In *Public Key Cryptography (1)*, volume 10769 of *Lecture Notes in Computer Science*, pages 315–347. Springer, 2018.
- [126] Slava Galperin, Dr. Carlisle Adams, Michael Myers, Rich Ankney, and Ambarish N. Malpani. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560, June 1999.

- [127] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
- [128] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM Conference on Computer and Communications Security*, pages 1179–1194. ACM, 2018.
- [129] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 1996.
- [130] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1999.
- [131] Rosario Gennaro, Michael O Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography.
- [132] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111. ACM, 1998.
- [133] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68. ACM, 2017.
- [134] github.com/alican. A Genetic Algorithm for Predicting Protein Folding in the 2D HP Model. <https://github.com/alican/GeneticAlgorithm>. Accessed: 2016-11-11.
- [135] Gideon Greenspan. Why many smart contract use cases are simply impossible. *CoinDesk*, Apr 2016.
- [136] Gridcoin. Gridcoin. <https://web.archive.org/web/20161013081149/http://www.gridcoin.us/>, 2016.
- [137] Gridcoin. Gridcoin (grc)–first coin utilizing Boinc–official thread. <https://web.archive.org/web/20160909032618/https://bitcointalk.org/index.php?topic=324118.0>, 2016.

- [138] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.
- [139] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.
- [140] Dick Grune and Criel J. H. Jacobs. *Parsing Techniques - A Practical Guide*. Monographs in Computer Science. Springer, 2008.
- [141] Ibrahim Hajjeh and Mohamad Badra. TLS Sign. Internet-Draft draft-hajjeh-tls-sign-04, Internet Engineering Task Force, December 2007. Work in Progress.
- [142] Carmit Hazay and Yehuda Lindell. A note on zero-knowledge proofs of knowledge and the ZKPOK ideal functionality. Cryptology ePrint Archive, Report 2010/552, 2010. <https://eprint.iacr.org/2010/552>.
- [143] Amir Herzberg. Folklore, practice and theory of robust combiners. *Journal of Computer Security*, 17(2):159–189, 2009.
- [144] Amir Herzberg, Markus Jakobsson, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive public key and signature systems. In *ACM Conference on Computer and Communications Security*, pages 100–110. ACM, 1997.
- [145] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 1995.
- [146] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*, page 11. ACM, 2013.
- [147] Marcia Hofmann. Court: Violating Terms of Service Is Not a Crime, But Bypassing Technical Barriers Might Be. *Electronic Frontier Foundation*, Jul 2010.
- [148] Neil Howe. A special price just for you. *Forbes*, Nov 2017.

- [149] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, pages 533–549. USENIX Association, 2016.
- [150] Intel Corporation. *Intel(R) Software Guard Extensions Programming Reference*, 329298-002us edition, 2014.
- [151] Intel Corporation. Intel(R) Software Guard Extensions Evaluation SDK User’s Guide for Windows* OS. <https://software.intel.com/sites/products/sgx-sdk-users-guide-windows>, 2015.
- [152] Intel Corporation. Intel(R) Software Guard Extensions SDK. <https://software.intel.com/en-us/sgx-sdk>, 2015.
- [153] Intel Corporation. Intel SGX for Linux. <https://01.org/intel-softwareguard-extensions>, 2016.
- [154] Intel Corporation. Intel Software Guard Extensions: Intel Attestation Service API. https://software.intel.com/sites/default/files/managed/3d/c8/IAS_1_0_API_spec_1_1_Final.pdf, 2016.
- [155] Intel Corporation. Public Key for Intel Attestation Service. <https://software.intel.com/en-us/sgx/resource-library>, 2016.
- [156] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30. ACM, 2007.
- [157] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, volume 152 of *IFIP Conference Proceedings*, pages 258–272. Kluwer, 1999.
- [158] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 1996.
- [159] Ari Juels, Lorenz Breidenbach, Alex Coventry, Sergey Nazarov, and Steve Ellis. Mixicles: Private decentralized finance made simple. Technical report, 2019. Chainlink whitepaper.
- [160] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of gyges: Investigating

the future of criminal smart contracts. In *ACM Conference on Computer and Communications Security*, pages 283–295. ACM, 2016.

- [161] Kames. The basics of decentralized identity. *uPort (Medium)*, Jul 2018.
- [162] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.
- [163] A. Kelkar, J. Bernard, S. Joshi, S. Premkumar, and E. G. Sirer. Virtual Notary. <http://virtual-notary.org/>, 2016.
- [164] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. *IACR Cryptol. ePrint Arch.*, 2020:269, 2020.
- [165] George Khoury. Violation of a website’s terms of service is not criminal. *Findlaw blog post*, 24 Jan. 2018.
- [166] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.
- [167] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. <https://web.archive.org/web/20160307052339/http://primecoin.org/static/primecoin-paper.pdf>, 2013.
- [168] Sunny King and Scott Nadal. PPcoin: Peer-to-peer crypto-currency with proof-of-stake. <https://web.archive.org/web/20161025145347/https://peercoin.net/assets/paper/peercoin-paper.pdf>, 2012.
- [169] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Auditable sharing of private data over blockchains. *Cryptology ePrint Archive*, 2018.
- [170] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium*, pages 279–296. USENIX Association, 2016.

- [171] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society, 2016.
- [172] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 944–961. IEEE Computer Society, 2018.
- [173] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [174] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *EuroSys*, pages 38:1–38:16. ACM, 2020.
- [175] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Financial Cryptography*, volume 8975 of *Lecture Notes in Computer Science*, pages 528–547. Springer, 2015.
- [176] Yanlin Li, Jonathan M. McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *USENIX Annual Technical Conference*, pages 409–420. USENIX Association, 2014.
- [177] Yehida Lindell. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005.
- [178] Litecoin Project. Litecoin, open source P2P digital currency. <https://litecoin.org>, retrieved Nov. 2014.
- [179] Eric Lombrozo, Johnson Lau, and Pieter Wuille. BIP141: Segregated witness (consensus layer). <https://web.archive.org/web/20160521104121/https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki>, 2015.
- [180] Haiyun Luo, Petros Zerfos, Jiejun Kong, Songwu Lu, and Lixia Zhang. Self-securing ad hoc wireless networks. In *ISCC*, pages 567–574. IEEE Computer Society, 2002.
- [181] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexan-

- der Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. *Cryptology ePrint Archive*, Report 2020/934, 2020. <https://eprint.iacr.org/2020/934>.
- [182] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: dynamic-committee proactive secret sharing. In *ACM Conference on Computer and Communications Security*, pages 2369–2386. ACM, 2019.
- [183] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. DelegaTEE: Brokered delegation using trusted execution environments. In *USENIX Security Symposium*, pages 1387–1403. USENIX Association, 2018.
- [184] David Mazieres. The Stellar consensus protocol: A federated model for Internet-level consensus. <https://web.archive.org/web/20161025142145/https://www.stellar.org/papers/stellar-consensus-protocol.pdf>, 2015.
- [185] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.
- [186] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *IEEE Symposium on Security and Privacy*, pages 475–490. IEEE Computer Society, 2014.
- [187] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of luck: an efficient blockchain consensus protocol. In *SysTEX@Middleware*, pages 2:1–2:6. ACM, 2016.
- [188] Kunihiro Miyazaki, Mitsuru Iwamura, Tsutomu Matsumoto, Ryôichi Sasaki, Hiroshi Yoshiura, Satoru Tezuka, and Hideki Imai. Digitally signed document sanitizing scheme with disclosure condition control. *IEICE Transactions*, 2005.
- [189] Amani Moin, Emin Gün Sirer, and Kevin Sekniqi. A Classification Framework for Stablecoin Designs. *arXiv*, Sep 2019.

- [190] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. <http://www.bitcoin.org/bitcoin.pdf>, 2008.
- [191] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *EuroS&P*, pages 305–320. IEEE, 2016.
- [192] Ventzislav Nikov and Svetla Nikova. On proactive secret sharing schemes. In *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2004.
- [193] Helen Nissenbaum. *Privacy in Context - Technology, Policy, and the Integrity of Social Life*. Stanford University Press, 2010.
- [194] John P. Njui. Coinbase custody service secures major institutional investor worth \$20 billion. *Ethereum World News*.
- [195] Mehrdad Nojoumian and Douglas R. Stinson. On dealer-free dynamic threshold schemes. *Adv. Math. Commun.*, 7(1):39–56, 2013.
- [196] Mehrdad Nojoumian, Douglas R. Stinson, and Morgan Grainger. Unconditionally secure social secret sharing scheme. *IET Information Security*, 4(4):202–211, 2010.
- [197] Justin O’Connell. Sophisticated trading bot exploits synthetix oracle, funds recovered. *Cointelegraph*, Jul 2019.
- [198] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 1st edition, 2009.
- [199] Andrew M. Odlyzko. Privacy, economics, and price discrimination on the internet. In *ICEC*, volume 50 of *ACM International Conference Proceeding Series*, pages 355–366. ACM, 2003.
- [200] @OpenLawOfficial. The future of derivatives: An end-to-end, legally enforceable option contract powered by Ethereum. *ConsenSys Media (Medium)*, Dec 2018.
- [201] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59. ACM, 1991.

- [202] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.
- [203] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *EUROCRYPT*, pages 643–673. Springer, 2017.
- [204] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *DISC*, volume 91 of *LIPICs*, pages 39:1–39:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- [205] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.
- [206] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.
- [207] Jack Peterson and Joseph Krug. Augur: a decentralized, open-source platform for prediction markets. *arXiv:1501.01042*, 2015.
- [208] Nathaniel Popper. Central banks consider Bitcoin’s technology, if not Bitcoin. *New York Times*, October 2016.
- [209] Giulio Prisco. Slock.it to introduce smart locks linked to smart ethereum contracts, decentralize the sharing economy. *Bitcoin Magazine*, 2015.
- [210] Parity Project. Transaction queue. <https://wiki.parity.io/Transactions-Queue>. Referenced on Nov 9 2018.
- [211] The Chromium Projects. The SPDY whitepaper. <https://dev.chromium.org/spdy/spdy-whitepaper>.
- [212] Bartosz Przydatek and Reto Strobil. Asynchronous proactive cryptosystems without agreement. In *ASIACRYPT*, pages 152–169. Springer, 2004.
- [213] Michael O. Rabin. Randomized byzantine generals. In *FOCS*, pages 403–409. IEEE Computer Society, 1983.

- [214] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.
- [215] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.
- [216] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. TLS-N: non-repudiation over TLS enabling ubiquitous content signing. In *NDSS*. The Internet Society, 2018.
- [217] Jeff John Roberts and Nicolas Rapp. Exclusive: Nearly 4 Million Bitcoins Lost Forever, New Study Says. *Fortune*, 11 2017.
- [218] Robert Ross and James Sewell. Foldingcoin white paper. <https://web.archive.org/web/20161022232226/http://foldingcoin.net/the-coin/white-paper/>, 2015.
- [219] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.
- [220] J. Salowey, A. Choudhury, and D. McGrew. AES Galois counter mode (GCM) cipher suites for TLS. RFC 5288, 2008.
- [221] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *Financial Cryptography*, volume 9603 of *Lecture Notes in Computer Science*, pages 515–532. Springer, 2016.
- [222] Nitesh Saxena, Gene Tsudik, and Jeong Hyun Yi. Efficient node admission for short-lived mobile ad hoc networks. In *ICNP*, pages 269–278. IEEE Computer Society, 2005.
- [223] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *ACM CCS*, 2011.
- [224] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: privacy-preserving outsourcing by distributed verifiable computation. In *ACNS*, 2016.
- [225] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? A study

- of the evolution of input validation vulnerabilities in web applications. In *Financial Cryptography*, 2011.
- [226] David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *ACM PODC*, 2008.
- [227] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. In *IEEE S&P*, 2015.
- [228] Andrew Sellars. Twenty years of web scraping and the computer fraud and abuse act. *BUJ Sci. & Tech. L.*, 24:372, 2018.
- [229] Adi Shamir. How to share a secret. *Communications of the ACM*, 1979.
- [230] Elaine Shi, Fan Zhang, Rafael Pass, Sridhar Devadas, Dawn Song, and Chang Liu. Trusted hardware: Life, the composable universe, and everything. Manuscript, 2015.
- [231] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank McKeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services, 2015.
- [232] Slashdot. Cornell researchers unveil a virtual notary. <https://tech.slashdot.org/story/13/06/20/1747249/cornell-researchers-unveil-a-virtual-notary>, June 2013.
- [233] Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin’s transaction processing. fast money grows on trees, not chains. In *Financial Cryptography*, Puerto Rico, 2015.
- [234] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. Content extraction signatures. In *International Conference on Information Security and Cryptology*, 2001.
- [235] Douglas R Stinson and Ruizhong Wei. Unconditionally secure proactive secret sharing scheme with combinatorial structures. In *SAC*, 1999.
- [236] SWIFT and Accenture. Swift on distributed ledger technologies. Technical report, SWIFT and Accenture, 2016.

- [237] Paul Syverson, R Dingledine, and N Mathewson. Tor: The second generation onion router. In *Usenix Security*, 2004.
- [238] Nick Szabo. Smart contracts. <http://szabo.best.vwh.net/smart.contracts.html>, 1994.
- [239] Tamir Tassa and Nira Dyn. Multipartite secret sharing by bivariate interpolation. *Journal of Cryptology*, 2009.
- [240] Emma Thompson and Coin Rivet. Three ways smart contracts are used in healthcare. *Yahoo! Finance*, May 2019.
- [241] K. Torpey. The conceptual godfather of augur thinks the project will fail. *CoinGecko*, 5 Aug. 2015.
- [242] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *EuroS&P*, pages 19–34. IEEE, 2017.
- [243] Jerry Useem. How online shopping makes suckers of us all. *The Atlantic*, Jul 2017.
- [244] User “QuantumMechanic”. Proof of stake instead of proof of work. <https://web.archive.org/web/20160320104715/https://bitcointalk.org/index.php?topic=27787.0>.
- [245] Angela Walch. Deconstructing ‘decentralization’: Exploring the core claim of crypto systems. *SSRN*, Feb 2019.
- [246] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [247] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS*, 2017.
- [248] Will Warren and Amir Bandehali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. *URL: <https://github.com/0xProject/whitepaper>*, 2017.
- [249] Theodore M Wong, Chenxi Wang, and Jeannette M Wing. Verifiable secret

- redistribution for archive systems. In *the 1st Security in Storage Workshop*, 2002.
- [250] Gavin Wood. Ethereum: A secure decentralized transaction ledger. <http://gavwood.com/paper.pdf>, 2014.
- [251] Jay J Wylie, Michael W Bigrigg, John D Strunk, Gregory R Ganger, Han Kiliccote, and Pradeep K Khosla. Survivable information storage systems. In *Computer*, 2000.
- [252] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Symp. Security and Privacy*, pages 640–656, May 2015.
- [253] Yuanzhong Xu, Weidong Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656, May 2015.
- [254] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, 1982.
- [255] Jeffrey Yasskin. Signed HTTP exchanges. Internet-Draft draft-yasskin-http-origin-signed-responses-05, 2019.
- [256] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society, 2009.
- [257] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *ACM Conference on Computer and Communications Security*, pages 270–282. ACM, 2016.
- [258] Fan Zhang, Philip Daian, Iddo Bentov, Ian Miers, and Ari Juels. Paralysis proofs: Secure dynamic access structures for cryptocurrency custody and more. In *AFT*, pages 1–15. ACM, 2019.
- [259] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert van Renesse. REM: resource-efficient mining for blockchains. In *USENIX Security Symposium*, pages 1427–1444. USENIX Association, 2017.
- [260] Fan Zhang, Sai Krishna Deepak Maram, Harjasleen Malvai, Steven

Goldfeder, and Ari Juels. DECO: liberating web data using decentralized oracles for TLS. In *ACM CCS*, 2020.

- [261] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. APSS: proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.
- [262] Frederik Zuiderveen Borgesius and Joost Poort. Online Price Discrimination and EU Data Privacy Law. *J. Consum. Policy*, 40(3):347–366, Sep 2017.
- [263] Guy Zyskind, Oz Nathan, and Alex Pentland. Decentralizing privacy: Using blockchain to protect personal data. In *IEEE Symposium on Security and Privacy Workshops*, pages 180–184. IEEE Computer Society, 2015.