COMPILERS FOR SECURE COMPUTATION

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

Rolph Recto

August 2023

COMPILERS FOR SECURE COMPUTATION

Rolph Recto, Ph.D.

Cornell University 2023

Sophisticated cryptographic mechanisms for *secure computation*, such as multi-party computation (MPC) and homomorphic encryption (HE), allow for computing over encrypted data. These mechanisms have the potential to be used in a vast array of applications, from joint computations between mutually distrusting parties to privacy-preserving offloading of computation to service providers. While scientific advances have brought the performance of these mechanisms closer into widespread practical use, they still remain the purview of experts because of their forbidding programming models. We argue that *compilers* are necessary to democratize secure computation: a compiler would allow developers can write applications without worrying about the complicated details of using cryptographic mechanisms such as MPC and HE.

To this end, we present two compilers for secure computation. First, we present Viaduct, an extensible compiler that can target a variety of cryptographic mechanisms. Developers write Viaduct programs in a security-typed language that allow them to annotate data with high-level security policies; the compiler then uses these annotations to determine which cryptographic mechanism can most efficiently and securely execute program components. Second, we present Viaduct-HE, a compiler that targets homomorphic encryption schemes. Modern HE schemes afford great performance improvements through *batching* many data elements into a single ciphertext, but data layouts that take advantage of batching most efficiently can be very complicated. Developers write Viaduct-HE programs in a high-level, array-oriented language; the compiler then searches for efficient strategies to lay out data in ciphertexts.

BIOGRAPHICAL SKETCH

Rolph Jester Javier Recto was born in Quezon City, Philippines on August 14, 1993 at 12:01 A.M. He was born after his identical twin brother, Ralph Jasper Javier Recto, who was born on August 14, 1993 at 12:00 A.M.

Rolph moved with his family to the United States in 2003. He graduated from Louisa County High School in spring 2012, and matriculated at the University of Virgnia in nearby Charlottesville in fall 2012. In his junior and senior years he worked as a research assistant for Professor Westley Weimer, which would solidify his interest in programming languages. He graduated from UVa with a double major in computer science and philosophy in spring 2016 and moved to New York City to start a job as a software engineer.

After deciding that he was tired of making a decent wage, he entered the computer science graduate program at Cornell in fall 2017.

There are countries out there where people speak English. But not like us—we have our own languages hidden in our carry-on luggage, in our cosmetics bags, only ever using English when we travel, and then only in foreign countries, to foreign people. It's hard to imagine, but English is their real language! Oftentimes their only language. They don't have anything to fall back on or to turn to in moments of doubt.

- Olga Tokarczuk, *Flights*

The secret in the poet's heart remains unknown to the secret police, despite their ability to predict his every thought, utterance, and movement by monitoring the cerebroscope which he must wear day and night. We can know which thoughts pass through a man's mind without understanding them. Our inviolable uniqueness lies in our poetic ability to say unique and obscure things, not in our ability to say obvious things to ourselves alone.

- Richard Rorty, *Philosophy and the Mirror of Nature*

ACKNOWLEDGEMENTS

As the old proverb goes for raising children, so it goes for writing a dissertation: it takes a village. There are too many people to thank for helping me in graduate school, but I'll take a shot here.

First and foremost, I would like to thank my advisor, Andrew Myers. Andrew has been a wonderful and supportive advisor throughout my time at Cornell. He has shaped how I think about computer science indelibly, and he has nurtured my work in innumerable ways. His intellectual breadth and voracity—many people have opinions about everything, but the man has *knowledgeable* opinions about everything—is something that I aspire to.

I would like to thank my collaborators—Coşku Acay, Max Algehed, Anitha Gollamudi, Tom Magrino, Mae Milano, Joshua Gancher, and Elaine Shi—for making research fun and less isolating. Needless to say, any contents of this dissertation you might find disagreeable are solely my fault and not theirs.

Thank you to the member of Applied Programming Langauges group (APL) for being a tight-knit group where I can bounce my research ideas—this includes Coşku Acay, Luke Bernick, Ethan Cecchetti, Vivian Ding, Suraaj Kanniwadi, Tom Magrino, Mae Milano, Haobin Ni, Silei Ren, Isaac Scheff, Charles Sherk, Ian Tomasik, Joshua Turcotti, Siqiu Yao, Yulun Yao, Drew Zagieboylo, and Yizhou Zhang. More broadly, thank you to the attendees of Cornell Programming Languages Discussion Group for fostering a community centered around PL research.

Thank you to the admin at Cornell CIS for helping me throughout the years. In particular, thank you to Vanessa Maley for helping with the organization of CIS Research Night and the Programming Languages Retreat.

Thank you to Sid Sanyam and Owen Arden for mentoring me during a wonderful internship at Meta, where I worked on a prototype information flow analysis for

Scala. My internship gave me a good perspective on what matters in industry and what it takes for research tools to be adopted in practice. Thank you also to my old friends Elizabeth Hilbert, Anish Tondwalkar, and Scott Newton for hanging out with me during my summer in California.

Thank you to the 14Strings! Rondalla for being a great group to play music with and helping me connect with my Filipino heritage. Special thanks to Jane and Cliff Maestro for organizing the group.

Thank you to my old friends at UVa Book Club: Nader Ahmed, Eric Chirtel, Greg Irving, Tori Gabriele, Sasan Mousavi, Kseniya Kenkeremath, Savannah Thieme, and Nick Shahbaz. I have enjoyed your company and have learned a lot from our many discussions.

Thank you to the many great friends I have met at Cornell, especially Pedro Azevedo de Amorim, Burcu Canacki, Niu Chen, Ela Correa, Ryan Doenges, Shiyi Li, Shir Maimon, Rachit Nigam, Michael Roberts, Goktug Saatcioglu, and Priya Srikumar. You all have made my grad school experience filled with fond memories. Special thanks to Jinglin Piao for being a great partner in the short time that we have known each other.

Thank you to my family for supporting me this whole time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Protecting the confidentiality and integrity of data *in transit*—i.e., data being communicated over a network—and data *at rest*—i.e., data in storage—has been the traditional raison d'être of cryptography. Mechanisms providing such protection are a ubiquitous feature of modern computing. For example, HTTPS, which uses Transport Layer Security (TLS) for encryption, is now the default protocol used in major web browsers to communicate with web servers [84]. Most operating systems have utilities for encrypting data in persistent storage (e.g., FileVault in macOS, cryptoloop and dm-crypt in GNU/Linux, BitLocker in Windows).

However, certain applications have stronger requirements that go beyond protecting data in transit and data at rest: they need to protect data *in use*. These applications need mechanisms for *secure computation*.[1]

1.1 The Need for Secure Computation

The need for secure computation arises primarily when the execution of an application is distributed between parties that do not fully trust each other. In such cases, one party might require that another party, though relied upon to perform some computation for the application, should not be able to acquire sensitive data or be able to unduly influence some output of the application. For example, an application might have party B perform some computation that requires A's private data. The application

[1] It is common to use *secure computation* to refer solely to the protection of the confidentiality of data in use, and to use *verifiable computation* to refer to the protection of the integrity of data in use. The term secure computation is also used sometimes to refer exclusively to a specific cryptographic mechanism, *multi-party computation* (MPC). Throughout this document, we use the term *secure computation* to mean broadly the protection of either the confidentiality or integrity of data in use, and never specifically to MPC.

might use standard cryptography like encryption to protect A's data en route to B from eavesdroppers. Once the data reaches B, however, encryption by itself falls short, since B must somehow perform computations over encrypted data, which is not generally possible without decryption keys (which we assume is not the case here because of the mutual distrust between A and B). Thus the application needs mechanisms for secure computation.

Applications requiring secure computation are becoming more common today; here we give some typical scenarios.

Cloud hosting. Instead of running applications on premises, many organizations choose to use the compute and storage of a cloud provider such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP) for cost savings or convenience. At the same time, clients want assurance that cloud providers are properly executing applications and are not exfiltrating sensitive data from their applications. Even if a client trusts the cloud provider, compute resources often are virtualized and host multiple tenants, allowing co-located tenants to mount attacks against an application [93].

Decentralized applications. Many systems are designed to be *decentralized* and to be run independently in a network crossing many administrative domains. This is especially the case for blockchains. In this setting, parties who run nodes in the network are not permissioned—i.e., parties can participate in the network freely without permission—and thus are mutually distrusting, with consensus reached by an established protocol (e.g., proof-of-work [81]). Since these networks are public, parties want to protect sensitive data (e.g., wallet amounts) while still allowing computations over such data.

	Trusted Hardware	Pure Cryptography
Pros	- simple programming model - efficient	- strong security guarantees - portable
Cons	- susceptible to side channel attacks - not portable	- inefficient - hard to program

Table 1.1: Trade-offs between secure computation mechanisms.

Third-party services. Organizations who have users of services provided by third parties might be wary of such services having access to proprietary data. Recently, Samsung banned employees from using ChatGPT and other generative AI services after finding confidential company data in a leak [52]. Individuals might also be reticent to upload sensitive personal data (e.g., health data, finance data) to such services.

Secure aggregations. Members of a group might find the need to compute some aggregated information about the group but want to keep their individual information secret. For example, hospitals might want to share aggregated patient information with each other, but be wary of releasing sensitive health information about individual patients.

1.2 Two Approaches to Secure Computation

Mechanisms for secure computation fall broadly into two camps. First, *trusted execution environments* (TEEs) such as Intel Software Guard Extensions (SGX) [28] and Sanctum [29] have special hardware modules (usually called *enclaves*) that allow applications to run securely even within a possibly compromised machine. Processors compute data "in the clear" as normal, but data resident in memory is decrypted / encrypted as needed. This provides strong isolation guarantees for applications running

in enclaves, ensuring that even a malicious operating system cannot exfiltrate sensitive data from the application. TEEs also have *remote attestation* mechanisms that assure remote clients that the application they intend to interact with is the one actually running on an enclave, ensuring the integrity of the application's outputs.

The second class of mechanisms for secure computation are purely cryptographic. Here we focus on mechanisms distinguished by their ability to support *general* computation. There are also many cryptographic mechanisms that target specific applications such as private set intersection [78]; we will not discuss these further.

Multi-party computation (MPC). MPC protocols allow several parties to jointly compute a function together over their private inputs. The protocol allows the parties to reveal the output of the function to each other without revealing anything else about their inputs.

Zero-knowledge proofs (ZKP). Zero-knowledge proofs in general allow one party (the *prover*) to provide an attestation that some statement is true without revealing any other information, allowing another party (the *verifier*) to believe the statement without gaining undue knowledge. ZKP protocols can be specifically used by the verifier to check that the result of a computation locally performed by the prover is correct. Crucially, the computation can depend on the prover's secret inputs, and the ZKP mechanism does not leak anything else about the input except what can be learned from the result.

Homomorphic encryption (HE). Homomorphic encryption schemes allow computation directly over ciphertexts. HE schemes are *homomorphic* in that the ciphertext operations correspond to plaintext operations: given messages m_1 and m_2 encrypted

to ciphertexts c_1 and c_2, the ciphertext c_3 that results from adding c_1 and c_2 should decrypt to the message $m_1 + m_2$. This allows a party who has access to ciphertext containing another party's private data to perform computations *without knowing the actual value of the data.* If the example application above used a homomorphic encryption scheme, party B could securely perform computations over an encrypted copy of party A's data.

1.2.1 Trade-offs

Neither of the two classes of secure computation mechanisms are preferable over the other in all scenarios. The two classes have different trade-offs, which is summarized by Table 1.1.

In general, because trusted hardware performs computations "in the clear," it has better performance and can support larger applications than purely cryptographic mechanisms. Its programming model is also simple, as legacy applications can be automatically adapted to run under TEEs [100, 8, 9]. At the same time, trusted hardware mechanisms have some drawbacks. Trusted hardware ties applications to a particular hardware architecture, which can be out of scope in certain settings (e.g., military applications might need to run on government-whitelisted hardware). Portability thus becomes an issue: applications have to be developed against a specific trusted hardware API, and adding support for new architectures would require rewriting parts of the application.

The biggest drawback to trusted hardware, however, is that its security guarantees can be undermined by vulnerabilities due to unforeseen design and implementation flaws. This holds especially true for Intel SGX, as there have been many attacks target-

ing it since its introduction [82]. These attacks range from exfiltrating secret data from an enclave through a variety of side channels (timing, microarchitectural state, memory access patterns), corruption of enclave computations, and more. Mitigations have been developed for these vulnerabilities, but their sheer number and wide range can severely undermine the assurance that trusted hardware's security guarantees actually hold.

Purely cryptographic mechanisms, meanwhile, are usually considered to be more secure than trusted hardware mechanisms. As standard in cryptography, the security of such mechanisms rests on the assumption that certain mathematical problems are hard to solve, and that the mechanisms have been properly implemented—for many, an easier thing to believe than the assumption that the complex trusted hardware designs of Intel or other hardware vendors are secure, and that such designs have been implemented properly. Purely cryptographic mechanisms are also not tied to specific hardware, and thus can easily be ported across many different architectures.

At the same time, developing applications using purely cryptographic mechanisms for secure computation is often much more challenging than developing applications that use trusted hardware.

First, purely cryptographic mechanisms are less efficient than trusted hardware; particularly for MPC and HE, the performance hit compared to computation "in the clear" can be many orders of magnitude. While algorithmic advances and hardware acceleration can narrow the gap, this performance hit often times imposes a hard limit on the size of applications that can be developed using purely cryptographic mechanisms.

Another difficulty in developing applications that use purely cryptographic mech-

anisms for secure computation is the relative lack of *programmability* of these mechanisms. The use of these mechanisms often requires expressing computation in unintuitive ways, and requires cryptographic expertise to develop efficient applications. The following highlights some of the major issues.

Matching security requirements and guarantees. Cryptographic mechanisms vary in their computational capabilities, their threat models, and their security guarantees. For example, a semi-honest MPC protocol cannot provide guarantees when parties deviate from the protocol, and zero-knowledge proofs can only attest the results of computations that depend on the prover's secret data. At the same time, the security requirements of a program component have a complex interplay with the trust assumptions of the parties executing the program, the dependency of the component to other parts of the program, and more. Determining whether a cryptographic mechanism can securely implement a program component—and whether the mechanism is the most efficient way to implement the component—thus becomes a vexed affair.

Circuit representation of programs. Purely cryptographic mechanisms for secure computation for the most part have a limited programming model that forces developers to write their programs in straight-line or "circuit" form. When writing applications directly against cryptographic libraries, this circuit requirement precludes the use of standard program constructs such as conditionals, loops, and dynamic memory accesses. Developers must then transform programs by unrolling loops, multiplexing ("muxing") conditionals, and so on.

Parameter selection. Cryptographic mechanisms often have parameters (e.g., key length) that modulate their guaranteed security level and efficiency. Parameter selec-

tion is especially complicated for lattice-based homomorphic encryption schemes—the family of homomorphic encryption schemes most commonly used today—wherein the efficiency of homomorphic operations and amount of computation supported by the scheme is extremely sensitive to the encryption parameters initially set. Setting parameters to ensure both security and efficiency thus requires significant expertise.

Domain-specific optimizations. Some optimizations apply only to a particular cryptographic mechanism. For example, some MPC protocols can be "mixed" together, allowing computations to be partitioned into subcomputations that are each computed by a different protocol [37]. If properly done, the partitioning can place computations into the protocol that can most efficiently implement them. However, computing efficient partitionings can be difficult [57]. As another example, homomorphic encryption schemes often support vectorization, allowing thousands of data elements to be encrypted in the same ciphertext. Thus programmers must be aware of such domain-specific optimizations and have cryptographic expertise to write programs with good performance.

1.3 Compilers for Secure Computation

To alleviate the difficulty of writing programs that use secure computation mechanisms, much prior work have focused on the development of *compilers* that target such mechanisms [55, 72, 2, 89, 18, 24, 103, 32, 22, 34, 68, 6, 31, 104, 74, 85, 30]. The idea is that instead of using a secure computation library directly, developers can instead write their programs in a high-level language and the compiler will do the hard work of translating the program into an implementation that directly uses the APIs of secure computation libraries. Instead of application developers shouldering the pro-

grammability burdens discussed earlier—converting programs to circuits [44, 83] parameter selection [34], domain-specific optimization [31, 104, 74, 57, 18]—these are instead handled automatically by the compiler.

In this dissertation, we discuss the design and implementation of compilers that tackle difficult programmability challenges for secure computation that still remain largely unresolved.

1.3.1 Extensibility

Existing compilers for secure computation compile to a single cryptographic mechanism—i.e., an MPC compiler will not generate code that uses ZKP, and vice versa. Because these compilers support only one mechanism with a fixed security guarantee, these compilers inherently assume that the security requirements of programs are uniform. But programs naturally have *heterogenous* security requirements: one part of a program might have a different security requirement than another part.

For instance, consider an "interval" program between three parties A, B, and C. A and B trust each other to not cheat, but neither one wants to reveal their private information to the other; meanwhile, C does not trust neither A or B and does not want to reveal its private information to them, and vice versa. The interval program computes the smallest interval over which the set of secret points owned by A and B lie; the program then checks whether a single point owned by C lies inside this interval. The security policy for the interval program requires that the secret points from A, B and C be kept secret, and that the only information publically revealed should be the extent of the interval and whether C's point lies within it. The first part of the program (computing the interval of points from A and B) can most naturally be implemented as a

multi-party computation between A and B; the second part of the program (computing whether C's point lies within the interval) can most naturally be implemented by C sending a zero-knowledge proof to A and B. Importantly, a traditional MPC or ZKP compiler cannot generate this implementation of the interval program, since it uses both MPC and ZKP.

A compiler for secure computation that reflects the heterogenous security requirements thus would be *extensible*, allowing support a variety of secure computation mechanisms. Additionally, the ability of such a compiler to automatically match program components with mechanisms that can implement them securely and efficiently would greatly ease application development, allowing programmers to develop secure distributed programs without expertise in secure computation.

To this end, we develop Viaduct, an extensible compiler for secure computation. Viaduct allows developers to write programs annotated with *information flow labels* [38, 80, 7, 96] that define their intended security policy. The compiler then analyzes these annotations to determine the most efficient cryptographic mechanism that can securely implement program components.

A major contribution of Viaduct is to extend information flow labels, historically used to specify security policies, to capture both the security policies of programs *and* the security guarantees of cryptographic mechanisms. With the uniform abstraction of labels, the compiler can easily match security requirements of program components with security guarantees of secure computation mechanisms. Along with labels, Viaduct has a set of well-defined extension points to allow developers to easily add new cryptographic mechanisms and allow the compiler to reason about the cost and computational constraints of such mechanisms. The compiler can thus naturally handle heterogenous security policies, such that of the interval program.

1.3.2 Vectorized HE

Modern homomorphic encryption schemes support SIMD computations. Because HE operations are still orders of magnitude slower than their cleartext counterparts, making good use of this SIMD capability by vectorizing HE programs is a particularly important optimization, as vectorization can drastically reduce the number of HE operations needed to be executed. At the same time, vectorized HE programs look very different from regular programs, requiring significant expertise to develop. A large literature of prior have developed expert-written vectorized HE programs [58, 47, 17, 3, 62], but automatic vectorization of arbitrary HE programs have only recently been explored [31, 104, 74].

Automatic vectorization for HE is significantly different from other vectorization regimes, such as compiler support for generating SIMD instructions in modern ISAs. In particular, vectorized HE has two main constraints that make it novel: (1) very wide vector widths, on the order of thousands of slots; and (2) limited data movement operations that preclude arbitrary shuffling of data in vectors. This makes the substantial literature on superword-level parallelism (SLP) vectorization [65, 76] and loop-nest vectorization [14] difficult to adapt to this setting, and requires new techniques for the automatic vectorization of HE programs.

To lower the burden of developing efficiently vectorized HE programs, we develop Viaduct-HE, a vectorizing compiler for homomorphic encryption. The key design feature that allows Viaduct-HE to generate efficient HE programs is its *array-oriented* source language. This allows the compiler to reason about computations at a much higher level of abstraction than reasoning about individual operations (i.e., at the level of arithmetic circuits). With source programs consisting of high-level array operations,

the compiler can give a simple representation to the layout of data in ciphertexts, which allows it to quickly search for efficient data layouts.

Together, the Viaduct and Viaduct-HE compilers greatly lower the programmability burden of cryptographic mechanisms for secure computation. This democratizes secure computation, making it a much more easily deployable and appealing technology to satisfy the security requirements of applications.

1.4 Roadmap

The rest of this dissertation is as follows. Chapter 2 discusses the design and implementation of the Viaduct compiler, as well as an evaluation that shows that it is a feasible approach to developing an extensible compiler for secure computation. Chapter 3 discusses the design and implementation of the Viaduct-HE compiler, as well as a an evaluation that shows that the prototype implementation can generate efficient vectorized HE programs that match expert-written HE programs. Finally, Chapter 4 concludes with a summary of the results of this dissertation, and a discussion of possible future research directions to extend Viaduct and Viaduct-HE.

CHAPTER 2

VIADUCT: AN EXTENSIBLE COMPILER FOR SECURE COMPUTATION

Modern distributed applications such as federated systems and decentralized blockchains typically involve parties from multiple administrative domains each with its own security policy. Companies might be required by law (such as the European Union's GDPR [45]) to protect user privacy when they process user data or share it with other companies. The lack of full trust among parties makes it difficult to develop such systems, especially when the security requirements necessitate the use of cryptographic mechanisms. Recent efforts from the cryptography community have pushed these mechanisms from theory to practical deployment [13], but a gap remains: they still require too much expertise to use successfully [39, 46, 36].

In this chapter we discuss Viaduct, a system that makes it easier for non-expert programmers to develop secure distributed programs that employ cryptography. It puts a variety of sophisticated cryptographic mechanisms in the hands of developers, including secure multiparty computation (MPC) protocols, zero-knowledge proofs (ZKP), and commitment schemes. Viaduct's *security-typed* language allows developers to annotate programs with information-flow labels to specify fine-grained security policies regarding the confidentiality and integrity of data and computation. An inference algorithm allows these annotations to be lightweight, and enables Viaduct to reject inherently insecure programs. Viaduct then enforces these policies by compiling high-level source code to secure distributed programs, automatically choosing efficient use of cryptography without sacrificing security. The compiler supports a range of cryptographic protocols whose security guarantees are characterized using information-flow labels. New protocols can be added to Viaduct by specifying their security properties and by implementing well-defined interfaces.

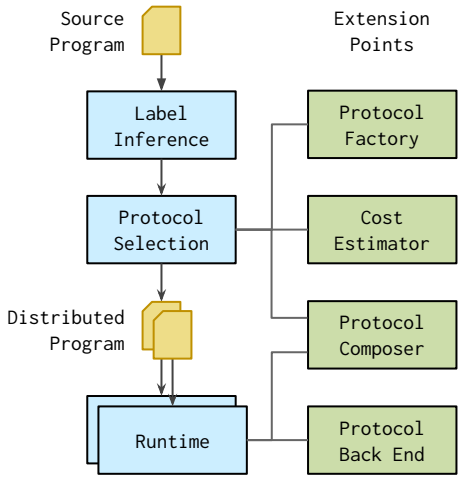

Figure 2.1: Architecture of Viaduct.

Although prior efforts have attempted to bridge this gap, most existing work focuses on compiling programs to a fixed set of cryptographic mechanisms. For example, some focus on compiling programs to MPC (e.g., Wysteria [89], ObliVM [72], SCALE-MAMBA [5]); others focus on ZKP (e.g., Pinocchio [85], Buffet [106], xjSNARK [64]). To our knowledge, by providing a unified abstraction to both specify security policies of programs and to specify security guarantees of cryptographic mechanisms, Viaduct is the first system to compile secure, distributed programs with an *extensible* suite of cryptography.

2.1 Overview

Figure 2.1 gives a high-level overview of Viaduct. Its compiler takes a high-level source program partially annotated with information-flow labels. The compiler infers labels consistent with programmer-supplied annotations to determine security requirements for all program components. Then for each component the compiler selects a protocol that matches these requirements, guiding the selection with a cost model. The output

```
host alice: {A ∧ B←}
host bob  : {B ∧ A←}

val a1, a2, a3 = input int alice
val b1, b2, b3 = input int bob
val a = min(a1, a2, a3)
val b = min(b1, b2, b3)
val b_richer = declassify a < b to {A ⊓ B}
output b_richer to alice, bob
```

Figure 2.2: Implementation of the historical millionaires' problem in Viaduct. Viaduct uses MPC for the comparison `a < b`, but computes the minima locally.

```
host alice: {A}
host bob  : {B}

val n: {B ∧ A←} =
  endorse (input int bob) from {B}
var tries: {A ⊓ B} = 5
var win: {A ⊓ B} = false
while (0 < tries ∧ !win) {
  val guess =
    declassify (input int alice) to {A ⊓ B→}
  val tguess: {A ⊓ B} =
    endorse guess from {A ⊓ B→}
  win = declassify (n == tguess) to {A ⊓ B}
  tries -= 1
}
output win to alice, bob
```

Figure 2.3: Guessing game, where Alice attempts to guess Bob's secret number. Viaduct uses zero-knowledge proofs so Alice learns nothing more than whether her guesses are correct. Most labels in this code can be inferred automatically.

is a secure and efficient distributed program, which hosts execute using the Viaduct runtime system. The Viaduct architecture has a small set of well-defined extension points, allowing developers to add support for new protocols with relative ease.

We give two examples to motivate and describe the Viaduct compilation process.

Historical Millionaires' Problem. Our first example is a slightly modified version of the "millionaires' problem" [108]. As in the classic formulation, two individuals, Alice and Bob, want to determine who has more money without revealing how much money they have to the other person. Rather than comparing their current wealth, in our "historical" variant Alice and Bob want to see who was richer at their *poorest*. Figure 2.2 shows an implementation of the historical millionaires' problem in Viaduct. The program compares Alice's lowest wealth with Bob's, and outputs the answer (`b_richer`) to both Alice and Bob.

Viaduct programs must specify the *hosts* that participate in the program, along with the *authority* that each host has, as shown in sections 2.1 to 2.1. All security policies in Viaduct are represented using *security labels* (in blue), which are defined formally in §2.1.1. Security labels capture both *confidentiality* and *integrity*. For example, host `alice` is given label $A \wedge B^{\leftarrow}$. Here, $B^{\leftarrow}$ is the *integrity component* of B (similarly, $B^{\rightarrow}$ is the confidentiality component of B). This label means that Alice fully trusts host `alice` (with both confidentiality and integrity), while Bob trusts host `alice` to execute the program correctly, but does not trust the host with his secret data.

All variables and expressions in Viaduct carry a security label, which is derived from the possible flows of information in the program. The variables in sections 2.1 to 2.1 carry the same label as their respective hosts, since they only involve data local to that host. However, the comparison `a < b` involves *both* hosts' private data, so has the higher security label $A \wedge B$. This label corresponds to data that is secret to and trusted by both principals. Since $A \wedge B$ corresponds to secret data, we require an explicit *declassification* to the label $A \sqcap B$, which describes data that both hosts can see and trust.

During protocol selection (§2.3), Viaduct chooses cryptographic protocols to se-

curely and efficiently execute our example. The central idea that allows Viaduct to select protocols automatically is that the security guarantees of protocols can also be captured by labels. Neither Alice nor Bob alone has enough authority to be responsible for the comparison, so Viaduct generates the following distributed implementation: Alice and Bob compute their respective minima locally but perform the comparison `a < b` in semi-honest MPC. A semi-honest MPC protocol works here because the authority labels assigned to the hosts indicate that Alice and Bob trust each other's hosts for integrity. Without that assumption, Viaduct is instead forced to select another protocol such as maliciously secure MPC.

There are typically multiple ways to assign protocols to a given program expression. For example, the computation of Alice's minimum on §2.1 could be securely performed in MPC, but since the computation requires the authority of Alice alone, it is cheaper yet still secure to do the computation locally on Alice's machine. Using its cost estimator, Viaduct compiles the optimal program described above.

After protocol selection, Viaduct outputs a distributed program which captures the required cryptography to execute the source program. Hosts can execute this distributed program using Viaduct's runtime system.

Guessing Game. Figure 2.3 presents a contrasting example. Here, Alice and Bob have security labels A and B respectively, modeling a *malicious* corruption scenario. Since they do not trust each other to execute the program correctly, semi-honest MPC is not applicable. Bob inputs a number `n`, and Alice has five attempts to guess the number. Since Bob's input initially has label B, it must first be *endorsed* to the label $B \wedge A^{\leftarrow}$, raising integrity so that Bob cannot unilaterally modify the value. This endorsement requires a cryptographic mechanism to protect the integrity and secrecy of variable `n`

throughout program execution.

Viaduct synthesizes a program in which Bob commits to `n` so that its value remains secret to Alice but Bob cannot later lie about the committed value. The statement `n == tguess` is computed by having Bob send a zero-knowledge proof (ZKP) to Alice, so that Alice can trust the outcome but learns no additional information. All other variables are replicated in plaintext across the two hosts.

These examples show that Viaduct is general, as it treats protocols such as MPC and ZKP uniformly.

2.1.1 Specifying Security Policies

In Viaduct, security policies capture a notion of authority. Policies are represented by *principals*, formulas composed of conjunctions and disjunctions over a set of base principals $\{A, B, C, \ldots\}$ and two special principals **0** and **1**. Principal **0** represents maximal authority and corresponds to the conjunction of all base principals; principal **1** represents minimal authority and corresponds to the disjunction of all base principals. We distinguish authority over *confidentiality* and over *integrity*. The security requirements of information are thus characterized by *labels* consisting of pairs $\langle p_{\mathrm{c}}, p_{\mathrm{i}} \rangle$ of two principals p_{c} and p_{i}, for confidentiality and integrity respectively.

A conjunction of principals $p_1 \wedge p_2$ represents combined authority. For confidentiality, this means the principal is allowed to read data that p_1 may read and also data that p_2 may read. For integrity, the conjunction may influence data that p_1 may influence, and also data p_2 may influence. A disjunction $p_1 \vee p_2$ corresponds to common authority, which may read or influence exactly the data that either p_1 and p_2 may individually.

Principals carry a natural partial order based on their authority. We write $p_1 \Rightarrow p_2$ to mean p_1 "acts for", or is at least as trusted as, p_2. This relation coincides with logical implication: for example, $p_1 \wedge p_2 \Rightarrow p_1$ and $p_1 \Rightarrow p_1 \vee p_2$.

It is convenient to have syntax that works over both components of labels simultaneously. So, we extend $\mathbf{0}$, $\mathbf{1}$, $\wedge$, $\vee$, and $\Rightarrow$ pointwise, and write one principal to mean that the two components are the same. For example, the annotation `{A}` denotes the label $\langle A, A\rangle$. To talk about confidentiality and integrity separately, we use projections, writing $\ell^{\rightarrow}$ for the confidentiality projection of ℓ and $\ell^{\leftarrow}$ for its integrity. Thus, `{B ∧ A←}` expands to $\langle B, B \wedge A\rangle$, meaning Bob's sole confidentiality and the combined integrity of Alice and Bob. These projections are defined formally as follows:

$$\langle p_{\mathrm{c}}, p_{\mathrm{i}}\rangle^{\rightarrow} \triangleq \langle p_{\mathrm{c}}, \mathbf{1}\rangle \qquad\qquad \langle p_{\mathrm{c}}, p_{\mathrm{i}}\rangle^{\leftarrow} \triangleq \langle \mathbf{1}, p_{\mathrm{i}}\rangle.$$

The *reflection* operator [109] swaps the two components:

$$\mathbb{X}(\langle p_{\mathrm{c}}, p_{\mathrm{i}}\rangle) \triangleq \langle p_{\mathrm{i}}, p_{\mathrm{c}}\rangle.$$

Viaduct programs assign labels to hosts to indicate the amount of trust placed in them, but there are also labels on data. The important insight, borrowed from FLAM [7], is that the same set of labels can be used to talk about both authority and information flow. When placed on data, a label takes on an information flow interpretation, specifying the minimum authority required to read and influence that data. As in FLAM, standard operations from information flow literature can be reformulated in terms of authority:

$$\ell_1 \sqsubseteq \ell_2 \iff \ell_2^{\rightarrow} \Rightarrow \ell_1^{\rightarrow} \quad \text{and} \quad \ell_1^{\leftarrow} \Rightarrow \ell_2^{\leftarrow} \qquad \text{(flows to)}$$

$$\ell_1 \sqcup \ell_2 \triangleq (\ell_1 \wedge \ell_2)^{\rightarrow} \wedge (\ell_1 \vee \ell_2)^{\leftarrow} \qquad \text{(join)}$$

$$\ell_1 \sqcap \ell_2 \triangleq (\ell_1 \vee \ell_2)^{\rightarrow} \wedge (\ell_1 \wedge \ell_2)^{\leftarrow} \qquad \text{(meet)}$$

The flows-to relation $\ell_1 \sqsubseteq \ell_2$ orders information flow policies: it means label ℓ_1 is more permissive about the use of information than ℓ_2. The join $\ell_1 \sqcup \ell_2$ is more restrictive

than both ℓ_1 and ℓ_2, and the meet $\ell_1 \sqcap \ell_2$ is more permissive than either ℓ_1 or ℓ_2. The most restrictive label—that of completely secret, untrusted data—is $\mathbf{0}^{\rightarrow} = \langle \mathbf{0}, \mathbf{1} \rangle$, and the least restrictive (public, trusted data) is $\mathbf{0}^{\leftarrow} = \langle \mathbf{1}, \mathbf{0} \rangle$.

A key property of this model, inherited from FLAM, is that labels form a bounded distributive lattice under both $(\Rightarrow, \wedge, \vee, \mathbf{0}, \mathbf{1})$ and $(\sqsubseteq, \sqcap, \sqcup, \mathbf{0}^{\leftarrow}, \mathbf{0}^{\rightarrow})$. This property is important for inferring labels that were not explicitly given by the programmer (§2.2.2).

2.1.2 Threat Model

Compiled programs run in a distributed setting in which each host executes a single thread concurrently with other hosts. Hosts communicate via message passing over secure, private, asynchronous channels. There is no shared memory that spans multiple hosts. We assume the attacker cannot observe wall-clock timing. Additionally, we are not concerned with availability, so the attacker can halt execution at any time.

In the setting of Viaduct, there is no single notion of an attacker. For example, in the historical millionaires problem, neither Alice nor Bob fully trust the other. To Alice, Bob is a potential attacker; Alice expects her security requirements to be met as long as the behavior of Bob's (partially trusted) host is accurately described by the label assigned to it ($B \wedge A^{\leftarrow}$). Conversely, to Bob, Alice is a potential attacker. Hence, we are concerned with security versus all possible attackers.

We model the power of an attacker using a label. The attacker can read the data on a host if the confidentiality of the attacker label is at least as trusted as that of the host, and can change data and code on the host if the integrity of the attacker label is at least as trusted as that of the host. We do not consider unreasonable attack scenarios

in which a host has compromised integrity but still enforces confidentiality.[1]

For example, in the historical millionaires' problem, there are five interesting corruption scenarios: no corrupted hosts; `alice` has corrupted confidentiality; `bob` has corrupted confidentiality; both have corrupted confidentiality; or both `alice` and `bob` are fully corrupted. The full corruption of a single host is not possible because the hosts trust each other, so if the integrity of one is corrupted then the other's integrity must be corrupted also.

2.1.3 Label Inference

Viaduct selects a protocol for every piece of data and computation in the program based on their authority requirements, represented as labels. Intuitively, program components must be executed by protocols with enough authority to defend the confidentiality of host inputs and the integrity of host outputs. These authority requirements are captured formally by a type system (§2.2.1), and Viaduct uses a novel inference algorithm (§2.2.2) to compute for all program components the minimum-authority labels that still respect the information-flow constraints on the program.

The only required label annotations on Viaduct programs are the authority labels on host declarations and labels on declassify/endorse expressions—all labels on variables can be elided, making annotation burden low. As we show in our evaluation, in practice these required annotations are enough to capture programmer intent: minimally annotated programs compile to the same distributed programs as their fully annotated versions.

[1] The semi-honest and malicious threat models common in cryptography correspond to corrupting only hosts' confidentiality and corrupting both hosts' confidentiality and integrity respectively.

2.1.4 Protocol Selection

After label inference, Viaduct performs *protocol selection*, which assigns a protocol to compute and store each subexpression and variable. Protocols encompass storage and computation performed "in the clear" as well as cryptographic mechanisms such as commitments, MPC and zero-knowledge proofs.

Each protocol P carries an associated authority label $\mathbb{L}(P)$, which approximates the security guarantees the protocol provides. Given a program component with minimum authority requirement ℓ, protocol selection only assigns P to execute that component if $\mathbb{L}(P) \Rightarrow \ell$—that is, if P meets the authority requirement for the program component.

Intuitively, given a program s and protocol P, we may imagine an *ideal functionality* P^s (in the style of UC [20]) which executes the program fragments of s that are assigned to P. The fragments of s that are assigned to P may depend on the computational abilities of P. For example, if P is a commitment protocol, then P^s is only able to store values but not perform any computations. If P is an MPC protocol, then P^s can execute computations that can be translated into circuits—the standard interface for MPC implementations.

P^s guarantees that the storage and computation it performs are protected at label $\mathbb{L}(P)$. In particular, the adversary cannot observe storage or computation performed by P^s unless its confidentiality is at least $\mathbb{L}(P)$; dually, the adversary cannot influence storage or computation performed by P^s unless its integrity is at least $\mathbb{L}(P)$.

Examples of protocols and their corresponding authority labels are given in Table 2.1. Following the above intuition for the security of functionalities P^s, the authority label of protocols are determined to be the least authority required of the adversary

Protocol	**Authority label**
$\mathrm{Local}(h)$	$\mathbb{L}(h)$
$\mathrm{Replicated}(H)$	$\bigsqcap_{h \in H} \mathbb{L}(h)$
$\mathrm{Commitment}(h_\mathrm{p}, h_\mathrm{v})$	$\mathbb{L}(h_\mathrm{p}) \wedge \mathbb{L}(h_\mathrm{v})^{\leftarrow}$
$\mathrm{ZKP}(h_\mathrm{p}, h_\mathrm{v})$	$\mathbb{L}(h_\mathrm{p}) \wedge \mathbb{L}(h_\mathrm{v})^{\leftarrow}$
$\mathrm{MAL\text{-}MPC}(H)$	$\bigwedge_{h \in H} \mathbb{L}(h)$
$\mathrm{SH\text{-}MPC}(H)$	let $I = \bigvee_{h \in H} \mathbb{L}(h)^{\leftarrow}$ $\left(\mathbb{X}(I) \vee \bigwedge_{h \in H} \mathbb{L}(h)^{\rightarrow} \right) \wedge I$

Table 2.1: Example protocols and security labels that represent their authority.

to corrupt the protocol (in confidentiality or integrity). We explain the example protocols below:

$\mathrm{Local}(h)$. No cryptography is performed, and data is stored and computations performed on host h in the clear. It provides exactly the authority of h.

$\mathrm{Replicated}(H)$. Data and computations are replicated on all hosts in set H, and replicated data is checked for equality when necessary. This protocol provides confidentiality $\bigvee_{h \in H} \mathbb{L}(h)^{\rightarrow}$ since all hosts hold the plaintext value. It provides integrity $\bigwedge_{h \in H} \mathbb{L}(h)^{\leftarrow}$ since all hosts must corrupt their local values for the value to be globally corrupted. Together, these labels form the label $\bigsqcap_{h \in H} \mathbb{L}(h)$.

$\mathrm{Commitment}(h_\mathrm{p}, h_\mathrm{v})$. Data is stored on h_p and commitments are placed on h_v. Commitments are computationally inexpensive but usually no computations can be performed with them. Commitments increase integrity without sacrificing confidentiality. Its confidentiality is $\mathbb{L}(h_\mathrm{p})^{\rightarrow}$ since only h_p holds the plaintext value, while h_v only holds a commitment. Its integrity is $(\mathbb{L}(h_\mathrm{p}) \wedge \mathbb{L}(h_\mathrm{v}))^{\leftarrow}$ for the same reason as for replication.

$\mathrm{ZKP}(h_\mathrm{p}, h_\mathrm{v})$. A zero-knowledge proof protocol where h_p is the prover and h_v is the verifier. The prover computes over its private data and sends the result to the

verifier, along with a *proof* that attests the value computed is correct. The proof reveals nothing about the private data except what can be gleaned from the result itself. Zero-knowledge proofs provide the same authority as commitments, for essentially the same reason: the prover holds all secret information and performs all computation, while the verifier only holds information which allows it to believe in the correctness of the result, but nothing more.

MAL-MPC(H). A corrupt-majority, maliciously secure multiparty computation protocol [49, 21, 19] performed by hosts H. The protocol allows hosts to jointly perform a computation over their private inputs, keeping these inputs secret to the other hosts and revealing only the result. The label $\bigwedge_{h \in H} \mathbb{L}(h)$ reflects that the confidentiality (resp., integrity) of data computed in MPC is compromised only if *all* participating hosts have compromised confidentiality (resp. integrity).

SH-MPC(H). A corrupt-majority, semi-honest secure multiparty computation protocol performed by hosts H. While the combined authority label is complex, its confidentiality and integrity projections are easy to understand. The integrity is equal to $\bigvee_{h \in H} \mathbb{L}(h)^{\leftarrow}$, since the integrity of the MPC computation may be compromised if *any* host behaves maliciously. The confidentiality is equal to

$$\left(\bigvee_{h \in H} \mathbb{X}(\mathbb{L}(h)^{\leftarrow}) \right) \vee \left(\bigwedge_{h \in H} \mathbb{L}(h)^{\rightarrow} \right).$$

The first disjunct captures the fact that confidentiality guarantees are discarded if the integrity of any host is compromised. The second disjunct states that, if all hosts follow the protocol correctly, the adversary can only learn the state of intermediate MPC computations if all hosts have corrupted confidentiality. Overall, this means that in order to compromise confidentiality guarantees of semi-honest MPC, either the integrity of any host or the confidentiality of all hosts must be compromised.

Alice (a)

cleartext	MPC
(1) **val** a1, a2, a3 = **input int** **val** am = min(a1, a2, a3) **send** am **to** (MPC(a,b),a) (3) **val** res = **recv** (MPC(a,b),a) **output** res	(2) **val** t_am = **recv** (Local(a),a) **val** am = InputGate(t_am) **val** bm = DummyInputGate() **val** lt = LTGate(am, bm) **val** v = ExecuteCircuit(lt) **send** v **to** (Replicated(a,b),a)

Bob (b)

cleartext	MPC
(1) **val** a1, a2, a3 = **input int** **val** bm = min(a1, a2, a3) **send** bm **to** (MPC(a,b),a) (3) **val** res = **recv** (MPC(a,b),b) **output** res	(2) **val** am = DummyInputGate() **val** t_bm = **recv** (Local(b),b) **val** bm = InputGate(t_bm) **val** lt = LTGate(am, bm) **val** v = ExecuteCircuit(lt) **send** v **to** (Replicated(a,b),b)

Figure 2.4: Execution of the compiled distributed program for the historical millionaires' problem using a cleartext back end and an MPC back end. Sends and receives are over protocol–host pairs (P, h). These messages are processed by the back end for protocol P at host h.

In particular, for the historical millionaires' example, the label of $\mathrm{SH\text{-}MPC}(\texttt{alice}, \texttt{bob})$ is $A \wedge B$. This is because hosts `alice` and `bob` are both assumed to have the high integrity of $(A \wedge B)^{\leftarrow}$. If `alice` and `bob` only have their own integrity, however, then the label is computed to be $A \vee B$. The protocol only has enough authority to perform computations over data public to both hosts, and neither host trusts the result. Indeed, semi-honest MPC offers little to no benefit if any host has lower integrity than any other.

2.1.5 Runtime

Viaduct provides a modular runtime system for executing compiled distributed programs, implemented as an interpreter. All hosts run the interpreter with the same compiled program, which then executes each host's portion of the program. During execution, the interpreter calls out to back ends implementing the cryptographic mechanisms used in the program. Back ends translate computations in the source language into their cryptographic realizations. For instance, the back ends for MPC and ZKP in our implementation build a circuit representation of the program as it executes.

Protocol back ends can send data to and receive data from each other, supporting the composition of protocols. Source-level declassification and endorsement induce this communication. For example, in Figure 2.2 on §2.1, the computation `a < b` is declassified from label $A \wedge B$ to $A \sqcap B$. This declassification causes the MPC protocol between Alice and Bob to execute its stored circuit for this comparison, and to output the result in cleartext.

Figure 2.4 shows the execution of the program compiled by Viaduct for the historical millionaires' problem. The program runs as follows. (1) First, the cleartext back ends on Alice and Bob's machines receive input locally and compute their respective minima. The back ends send the minima as secret inputs to their respective MPC back ends, which create input gates for these inputs. (2) Next, the MPC back ends on Alice and Bob's machines each create an operation gate that compares Alice and Bob's secret inputs. The back ends jointly execute the circuit with the comparison result as output, which they send to their respective cleartext back ends. (3) Finally, the cleartext back ends on Alice and Bob's machines both receive from their MPC back ends and output the result.

$$
\begin{array}{lllll}
\text{Temporaries} & t & \in & \mathbb{T} \\
\text{Assignables} & x & \in & \mathbb{X} \\
\text{Hosts} & h & \in & \mathbb{H} \\
\text{Labels} & \ell & \in & \mathbb{L} \\
\\
\text{Base Types} & \beta & ::= & \textbf{unit} \mid \textbf{bool} \mid \textbf{int} \\
\text{Data Types} & D & ::= & \textbf{Cell}_{\beta} \mid \textbf{Array}_{\beta} \\
\\
\text{Values} & v & ::= & () \mid \textbf{true} \mid \textbf{false} \mid i \in \mathbb{Z} \\
\text{Unary Operators} & op_1 & ::= & \textbf{not} \mid - \mid \ldots \\
\text{Binary Operators} & op_2 & ::= & \wedge \mid \vee \mid + \mid \times \mid = \mid \ldots \\
\text{Methods} & m & ::= & \textbf{get} \mid \textbf{set} \mid \ldots \\
\\
\text{Atomic Expr.} & a & ::= & v \mid t \\
\text{Expressions} & e & ::= & \\
 & \multicolumn{3}{l}{\quad \mid a \mid op_n(a_1, \ldots, a_n) \mid x.m(a_1, \ldots, a_n)} \\
 & \multicolumn{3}{l}{\quad \mid \textbf{declassify } a \textbf{ to } \ell \mid \textbf{endorse } a \textbf{ from } \ell} \\
 & \multicolumn{3}{l}{\quad \mid \textbf{input}_{\beta}\ h \mid \textbf{output } a \textbf{ to } h} \\
\\
\text{Statements} & s & ::= & \\
 & \multicolumn{3}{l}{\quad \mid \textbf{let } t = e \textbf{ in } s \mid \textbf{new } x = D(a_1, \ldots, a_n) \textbf{ in } s} \\
 & \multicolumn{3}{l}{\quad \mid \textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2 \mid b : \textbf{loop } s \mid \textbf{break } b} \\
 & \multicolumn{3}{l}{\quad \mid s_1; s_2 \mid \textbf{skip}}
\end{array}
$$

Figure 2.5: Abstract syntax of Viaduct's source language.

2.2 Source Language

The syntax for Viaduct's source language, a simplified version of the *surface* language, is given in Figure 2.5. The language supports base types such as booleans and integers, along with their usual operators. Surface-level assignables (`val` and `var` declarations) and arrays are uniformly represented as *data types*, a restricted form of objects. Like regular objects, they are created using constructors (**new** declarations) and contain methods. For simplicity, we only include three data types: immutable/mutable cells, which model surface-level assignables, and arrays. Arrays are dynamically sized but statically allocated: the size of an array can depend on values known only at run time, but array references cannot be rebound to different names or stored in arrays.

We distinguish between fully evaluated atomic expressions a, and expressions e that evaluate to values and may have side effects. Methods include **get** and **set** operations for both mutable cells and arrays (for which they take an index as an extra argument). Input/output expressions allow programs to interact with hosts. The **declassify** expression marks locations where private data is explicitly allowed to flow to public data, while the **endorse** expression marks locations where untrusted data is explicitly allowed to influence trusted data.

Statements consist of let-bindings, assignable declarations, as well as the usual conditionals, loops, and sequential composition. Temporaries bind values while assignables bind instances of data types. We require all intermediate computations to be let-bound by a temporary, enforcing a variant of *A-normal form* [41]. We use the more general loop-until-break statements instead of the more traditional while loops, simplifying the conversion to A-normal form. A break statement ($\mathbf{break}\ b$) includes an identifier b that names the loop it breaks out of. While loops are recovered easily:

$$\mathbf{while}\ e\ \mathbf{do}\ s \triangleq b : \mathbf{loop}\ (\mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ \mathbf{break}\ b).$$

2.2.1 Label Checking

Viaduct's type system enforces secure information flow in a standard way. The type system serves two purposes. First, it helps programmers ensure there are no unintended information flows: secrets are not leaked to and data is not corrupted by unauthorized principals. Second, it specifies what labels can be assigned to variables and expressions that the user did not explicitly annotate.

Figure 2.6 presents label checking rules for expressions and selected statements. Expressions are checked by the judgment $\Gamma; pc \vdash e : \ell$, which means that e has la-

$$\boxed{\Gamma \vdash a : \ell} \qquad \boxed{\Gamma; pc \vdash e : \ell}$$

$$\frac{}{\Gamma \vdash v : \ell} \qquad \frac{\Gamma(t) = \ell_t \qquad \ell_t \sqsubseteq \ell}{\Gamma \vdash t : \ell}$$

$$\frac{\Gamma \vdash a_i : \ell}{\Gamma; pc \vdash op_n(a_1, \ldots, a_n) : \ell} \qquad \frac{\begin{array}{c}\Gamma(x) = \ell_x \qquad pc \sqsubseteq \ell_x \\ \Gamma \vdash a_i : \ell_x \qquad \ell_x \sqsubseteq \ell\end{array}}{\Gamma; pc \vdash x.m(a_1, \ldots, a_n) : \ell}$$

$$\frac{\begin{array}{c}pc \sqsubseteq \ell_{\mathrm{t}} \qquad \Gamma \vdash a : \ell_{\mathrm{f}} \qquad \ell_{\mathrm{f}}^{\leftarrow} = \ell_{\mathrm{t}}^{\leftarrow} \\ \ell_{\mathrm{f}}^{\rightarrow} \sqsubseteq \ell_{\mathrm{t}}^{\rightarrow} \sqcup \mathbb{X}(\ell_{\mathrm{f}}^{\leftarrow}) \qquad \ell_{\mathrm{t}} \sqsubseteq \ell\end{array}}{\Gamma; pc \vdash \textbf{declassify } a \textbf{ to } \ell_{\mathrm{t}} : \ell} \qquad \frac{\begin{array}{c}pc \sqsubseteq \ell_{\mathrm{t}} \qquad \Gamma \vdash a : \ell_{\mathrm{f}} \qquad \ell_{\mathrm{f}}^{\rightarrow} = \ell_{\mathrm{t}}^{\rightarrow} \\ \ell_{\mathrm{f}}^{\leftarrow} \sqsubseteq \ell_{\mathrm{t}}^{\leftarrow} \sqcup \mathbb{X}(\ell_{\mathrm{f}}^{\rightarrow}) \qquad \ell_{\mathrm{t}} \sqsubseteq \ell\end{array}}{\Gamma; pc \vdash \textbf{endorse } a \textbf{ from } \ell_{\mathrm{f}} : \ell}$$

$$\frac{pc \sqsubseteq \mathbb{L}(h) \qquad \mathbb{L}(h) \sqsubseteq \ell}{\Gamma; pc \vdash \textbf{input}_{\beta}\ h : \ell} \qquad \frac{pc \sqsubseteq \mathbb{L}(h) \qquad \Gamma \vdash a : \mathbb{L}(h)}{\Gamma; pc \vdash \textbf{output } a \textbf{ to } h : \ell}$$

$$\boxed{\Gamma; pc \vdash s} \qquad \frac{\Gamma; pc \vdash e : \ell \qquad pc \sqsubseteq \ell \qquad (\Gamma,\ t : \ell); pc \vdash s}{\Gamma; pc \vdash \textbf{let } t = e \textbf{ in } s} \qquad \frac{\Gamma \vdash a_i : \ell \qquad pc \sqsubseteq \ell \qquad (\Gamma,\ x : \ell); pc \vdash s}{\Gamma; pc \vdash \textbf{new } x = D(a_1, \ldots, a_n) \textbf{ in } s}$$

$$\frac{\begin{array}{c}pc \sqsubseteq pc' \qquad \Gamma \vdash a : pc' \\ \Gamma; pc' \vdash s_1 \qquad \Gamma; pc' \vdash s_2\end{array}}{\Gamma; pc \vdash \textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2} \qquad \frac{pc \sqsubseteq pc' \qquad (\Gamma,\ b : pc'); pc' \vdash s}{\Gamma; pc \vdash b : \textbf{loop } s}$$

$$\frac{\Gamma(b) = \ell_b \qquad pc \sqsubseteq \ell_b}{\Gamma; pc \vdash \textbf{break } b} \qquad \frac{\Gamma; pc \vdash s_1 \qquad \Gamma; pc \vdash s_2}{\Gamma; pc \vdash s_1; s_2} \qquad \frac{}{\Gamma; pc \vdash \textbf{skip}}$$

Figure 2.6: Information flow checking rules for expressions and statements.

bel ℓ under the context on the left. Here, Γ is a finite partial map from temporaries, assignables, or loop names to labels:

$$\text{Label Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma,\ t : \ell \mid \Gamma,\ x : \ell \mid \Gamma,\ b : \ell$$

The *program counter* label pc is a standard way to prevent implicit flows of information via control flow [95]. The rules for method calls and input/output expressions differ from those in standard security-typed languages in that they also include premises with pc checks. These checks are required because these expressions may induce com-

munication between hosts, and hosts may learn secrets based on which requests they receive. Prior work that targets the distributed setting contains similar checks to control *read channels* [112].

Statement checking rules have the form $\Gamma; pc \vdash s$; they are largely standard [95]. Because we assume attackers cannot observe timing nor analyze traffic, the rule for conditional statements does not require branches to have the same timing behavior or effects (e.g., method calls, input/output).

Nonmalleable Information Flow Control. Information flow type systems typically aim to enforce a compositional security property such as *noninterference* [48]. Noninterference is a strong property but it is too restrictive for practical applications, which usually have a more nuanced policy for secure information flow. Hence, like most languages supporting information flow control (e.g., [79, 87, 16]), Viaduct allows programmers to signify the exceptions to a noninterference policy through *downgrading* expressions.

Downgrading enables information flows that would violate noninterference, so it can be dangerous. This is especially true in the distributed setting, where storage and computation can be performed by hosts that one does not fully trust. Downgrading confidentiality (declassification) allows secret information to be treated as public information—a necessity for many applications, but doing so might allow a corrupted host to control when information is released or what information is released. Downgrading integrity (endorsement) allows untrusted information to be treated as trusted information, but might enable a corrupted host to trick an honest one into accepting mauled secrets.

The property of *nonmalleable information flow control* (NMIFC) [23] prevents both

of these abuses of downgrading by combining two properties: *robust declassification* [111] and *transparent endorsement* [23]. Robust declassification requires that principals to which data is declassified could not have influenced either the decision to declassify or the data itself. Meanwhile, transparent endorsement prevents trusting mauled secrets by ensuring that information can only be endorsed if the providing principal can read it.

The declassification and endorsement rules in Figure 2.6 enforce NMIFC using the reflection operator $\overline{\mathbb{X}}$ (§2.1.1). The rules prevent the program from downgrading information with *compromised labels* [109], in which confidentiality exceeds integrity. These rules generate authority requirements that prevent the Viaduct compiler from placing data and computation on insufficiently trustworthy hosts. For example, consider a program where a server releases secret information to a client when the client guesses the correct password:

```
host server: {S}, client: {1}
val info: int{S}, pw: int{S}, guess: int{1}
if (declassify (pw == guess) to {1})
  output (declassify info to {1}) to client
```

This program violates robust declassification, because the decision to declassify `info` depends on (low-integrity) `guess`. Without the restrictions on downgrading, Viaduct could compile the program to store the guard `pw == guess` (with label **1**) on the *client*. The client could simply claim to the server that its guess is correct! For this program to type-check with NMIFC, endorsement is needed to make the guard high-integrity. A naive programmer might think to endorse the entire guard, but this (nontransparent) endorsement could still be compiled in a way that lets an untrusted host supply its value. The correct solution is to explicitly endorse `guess` before declassifying the

$$\ell_1 \sqsubseteq \ell_2 \rightsquigarrow \mathrm{C}(\ell_2) \Rightarrow \mathrm{C}(\ell_1),\ \mathrm{I}(\ell_1) \Rightarrow \mathrm{I}(\ell_2)$$
$$\ell_{\mathrm{f}}^{\rightarrow} \sqsubseteq \ell_{\mathrm{t}}^{\rightarrow} \sqcup \overline{\mathrm{X}}(\ell_{\mathrm{f}}^{\leftarrow}) \rightsquigarrow \mathrm{I}(\ell_{\mathrm{f}}) \wedge \mathrm{C}(\ell_{\mathrm{t}}) \Rightarrow \mathrm{C}(\ell_{\mathrm{f}})$$
$$\ell_{\mathrm{f}}^{\leftarrow} \sqsubseteq \ell_{\mathrm{t}}^{\leftarrow} \sqcup \overline{\mathrm{X}}(\ell_{\mathrm{f}}^{\rightarrow}) \rightsquigarrow \mathrm{I}(\ell_{\mathrm{f}}) \Rightarrow \mathrm{C}(\ell_{\mathrm{f}}) \vee \mathrm{I}(\ell_{\mathrm{t}})$$

Figure 2.7: Translating flows-to constraints over labels to acts-for constraints over label components.

comparison; since `guess` is not secret, the endorsement is transparent. The resulting labels correctly force Viaduct to put the comparison on the server.

2.2.2 Label Inference

Checking secure information flow is not enough; for protocol selection, the compiler also needs the labels of all expressions. We present an algorithm to infer these labels.

As in prior work on inferring information flow labels [79, 87], information flow checking reduces to a system of flows-to ($\sqsubseteq$) constraints over label constants and label variables. Type inference collects these premises from Figure 2.6, and generates fresh label variables for labels that appear in a premise of a rule but not its conclusion (e.g., pc' in the rule for if statements). The inference algorithm finds a label-variable assignment that satisfies all the constraints, if possible.

The algorithm computes the *minimum-authority* solution, the choice of labels requiring the least amount of confidentiality and integrity for each component. Minimum-authority labels are desirable because higher authority is achieved only through more trust or costly cryptography.

First, we translate the flows-to ($\sqsubseteq$) constraints over *labels*, which appear in rule

Constraint	Update rule
$L_1 \Rightarrow L_2$	$L_1^{i+1} := L_1^i \wedge L_2^i$
$L_1 \wedge p_2 \Rightarrow L_3$	$L_1^{i+1} := L_1^i \wedge (p_2 \rightarrow L_3^i)$
$L_1 \Rightarrow L_2 \vee L_3$	$L_1^{i+1} := L_1^i \wedge (L_2^i \vee L_3^i)$

Table 2.2: Update rules for solving acts-for constraints.

premises, to acts-for ($\Rightarrow$) constraints over the underlying *label components* as shown in Figure 2.7. Here, $\mathrm{C}(\ell)$ and $\mathrm{I}(\ell)$ are functions that project the confidentiality and integrity components, respectively, of label ℓ. These components are constants p when the label is known, and variables L otherwise.

We then adapt the algorithm of Rehof and Mogensen [91] for iteratively solving semilattice constraints. All principal variables are initialized to $\mathbf{1}$ and unsatisfied constraints are used to update variables repeatedly, until a fixed point is reached, according to the rules in Table 2.2. Constraints of the form $L_1 \Rightarrow L_2$ or $L_1 \Rightarrow L_2 \vee L_3$ are used to perform the corresponding update.

However, the rules in Figure 2.7 can also generate constraints of the form $L_1 \wedge p_2 \Rightarrow L_3$, arising from the typing rule for robust declassification. The term p_2 is always a constant since Viaduct requires annotations on declassify operations, so the value of L_1 can be updated safely to $p_2 \rightarrow L_3$, which denotes the weakest authority p such that $p \wedge p_2 \Rightarrow L_3$. The label p is also known as the *relative pseudocomplement* of p_2 with respect to L_3. When a lattice supports the $\rightarrow$ operation, it is a *Heyting algebra* [94], allowing each update rule to lower the left-hand-side variable to the minimum authority satisfying the constraint. Any free distributive lattice, such as our lattice of principals, is a Heyting algebra.

We prove that the iterative analysis we use for label inference always terminates and computes the minimum-authority solution. First, we construct the $\rightarrow$ operator

over the lattice of principals, which occurs in the update rules.

Constructing the Relative Pseudo-Complement

We show that any free distributive lattice, like our lattice of principals, is a Heyting algebra, and thus the relative pseudocomplement operator ($\rightarrow$) we use in our label inference algorithm (§2.2.2) is well-defined. While this is a standard result in algebra, we believe it is illuminating to see the actual construction, as we use its concrete value to compute minimum-authority labels.

Free Distributive Lattices. Let P be an arbitrary set. The standard construction for the free distributive lattice over P takes finite sets of finite subsets of P as elements, which we write as

$$\{A_i\}_{i\in[n]} \quad (\text{where } A_i \subseteq P).$$

An element of this form is interpreted as a join of meets, that is, $\{A_i\}_{i\in[n]}$ intuitively stands for

$$\Big(\bigwedge A_1\Big) \vee \ldots \vee \Big(\bigwedge A_n\Big).$$

In addition to every A_i being finite, we require that there is no A_i and A_j such that $A_i \subseteq A_j$ for $i \neq j$ since this makes A_j redundant per our interpretation (i.e. $(\bigwedge A_i) \vee (\bigwedge A_j) = \bigwedge A_i$). We assume all such components are dropped implicitly.

Define

$$\{A_i\}_{i\in[n]} \vee \{B_j\}_{j\in[m]} = \{A_i\}_{i\in[n]} \cup \{B_j\}_{j\in[m]}$$

and

$$\{A_i\}_{i\in[n]} \wedge \{B_j\}_{j\in[m]} = \{A_i \cup B_j \mid i \in [n], j \in [m]\}.$$

It is straightforward to verify that these definitions satisfy the properties for being the join and the meet, respectively. It is also easy to see that

$$0 = \{\} \qquad \text{and} \qquad 1 = \{\{\}\}.$$

Finally, ordering can be derived in the standard way for distributive lattices:

$$A \leq B \iff A \vee B = B.$$

We find it useful to have a more direct definition, which we can derive by expanding the previous definition:

$$\{A_i\}_{i\in[n]} \leq \{B_j\}_{j\in[m]} \iff \forall i \in [n].\exists j \in [m].B_j \subseteq A_i.$$

Heyting Algebras. A Heyting algebra is a bounded distributive lattice where every inequality of the form

$$A \wedge X \leq B$$

has a greatest solution. This solution is named $A \to B$ to appeal to logical intuition as $A \to B$ is the weakest (i.e. the greatest) proposition such that $A \wedge (A \to B)$ logically implies B. We show that every free distributive lattice forms a Heyting algebra.

Define

$$\{A_i\}_{i\in[n]} \to \{B_j\}_{j\in[m]} = \bigwedge_{i\in[n]} \{B_j \setminus A_i \mid j \in [m]\}.$$

First, we claim this is in fact a solution to the above inequality, that is,

$$\{A_i\}_{i\in[i]} \wedge \bigwedge_{i\in[n]} \{B_j \setminus A_i \mid j \in [m]\} \leq \{B_j\}_{j\in[j]}.$$

Proof. By applying the definition of $\wedge$ repeatedly ($i + 1$ times), we can rewrite the left-hand side as

$$\{A_i \cup (B_{j_1} \setminus A_1) \cup \ldots \cup (B_{j_n} \setminus A_n) \mid i \in [n], j_1, \ldots, j_n \in [m]\}.$$

Using the direct definition of $\leq$ from before, it suffices to show that there exists $j \in [m]$ such that

$$B_j \subseteq A_i \cup (B_{j_1} \setminus A_1) \cup \ldots \cup (B_{j_n} \setminus A_n)$$

for all i, j_1, …, j_n. Picking $j = j_i$, we get

$$B_{j_i} \subseteq A_i \cup (B_{j_i} \setminus A_i) \subseteq A_i \cup (B_{j_1} \setminus A_1) \cup \ldots \cup (B_{j_n} \setminus A_n).$$

□

Next, we need to prove that this solution is the greatest. Assume there is an X such that $A \wedge X \leq B$ where $A = \{A_i\}_{i \in [n]}$, $B = \{B_j\}_{j \in [m]}$, and $X = \{X_k\}_{k \in [o]}$. Our goal is to show

$$\{X_k\}_{k \in [o]} \leq \bigwedge_{i \in [n]} \{B_j \setminus A_i \mid j \in [m]\}.$$

Proof. Using the universal property of $\wedge$ and the direct definition of $\leq$ from before, it is sufficient to prove

$$\forall i \in [n], k \in [o]. \exists j \in [m]. B_j \setminus A_i \subseteq X_k.$$

Let i and k be arbitrary. Since $\{A_i\} \leq A$ and $\{X_k\} \leq X$, we know

$$\begin{aligned} \{A_i\} \wedge \{X_k\} \leq A \wedge X \leq B &\implies \exists j \in [m]. B_j \subseteq A_i \cup X_k \\ &\implies \exists j. B_j \setminus A_i \subseteq X_k. \end{aligned}$$

□

Termination and Optimality of Label Inference

It is well-known that iterative analysis always terminates given that the function defined by the update rules is monotone, and that the lattice over which the algorithm runs is of finite height [61]. Because the update rules take the meet of the current solution with some other lattice element, it is immediate that the function is monotone. Because it is the free distributive lattice, all elements of the principal lattice can be represented in normal form as a join of meets of atomic principals, and thus is of finite size when the set of atomic principals is finite. Thus the principal lattice is of finite height as long as it is generated from a finite set of atomic principals. We know any program can only reference a finite set of unique atomic principals in its text since any program has a finite set of labels in its text, and each label can only mention a finite set of atomic principals. Thus for any program, the principal lattice is of finite height.

Finally, we show that the algorithm computes the optimal (minimum-authority) solution. It is also well-known by appeal to Kleene's fixed-point theorem that iterative analysis computes the greatest-fixpoint solution of a monotone function. Thus to prove optimality it is sufficient to show that any solution to the constraints must lower-bound the current solution computed from the update rules, and thus must lower-bound the greatest-fixpoint solution computed by the algorithm.

Proof. We prove the statement by induction over the number of iterations performed by iterative analysis. The base case is immediate since all principal variables are initialized to **1**, the top of the principal lattice.

To prove the inductive case, we perform a case analysis over the update rules:

Case 1: $L_1^{i+1} := L_1^i \wedge L_2^i$**.** This update rule is for constraint $L_1 \Rightarrow L_2$, and thus for any

solution ψ it must be the case that $\psi(L_1) \Rightarrow \psi(L_2)$. By the induction hypothesis, we know $\psi(L_1) \Rightarrow L_1^i$ and $\psi(L_2) \Rightarrow L_2^i$, and thus $\psi(L_1) \Rightarrow L_2^i$ by transitivity. Since $\wedge$ is the greatest lower bound, $\psi(L_1) \Rightarrow L_1^i \wedge L_2^i = L_1^{i+1}$, as needed.

Case 2: $L_1^{i+1} := L_1^i \wedge (p_2 \rightarrow L_3^i)$**.** This update rule is for $L_1 \wedge p_2 \Rightarrow L_3$, and thus for any solution ψ it must be the case that $\psi(L_1) \wedge p_2 \Rightarrow \psi(L_3)$. By the inductive hypothesis we know that $\psi(L_1) \Rightarrow L_1^i$ and $\psi(L_3) \Rightarrow L_3^i$, and thus $\psi(L_1) \wedge p_2 \Rightarrow L_3^i$ by transitivity. By definition we know $p_2 \rightarrow L_3^i$ is the greatest principal p such that $p \wedge p_2 \Rightarrow L_3^i$, so $\psi(L_1) \Rightarrow p_2 \rightarrow L_3^i$. Since $\wedge$ is the greatest lower bound, $\psi(L_1) \Rightarrow L_1^i \wedge (p_2 \rightarrow L_3^i) = L_1^{i+1}$ as needed.

Case 3: $L_1^{i+1} := L_1^i \wedge (L_2^i \vee L_3^i)$**.** This update rule is for constraint $L_1 \Rightarrow L_2 \vee L_3$, so for any solution ψ it must be the case that $\psi(L_1) \Rightarrow \psi(L_2) \vee \psi(L_3)$. By the inductive hypothesis we know that $\psi(L_1) \Rightarrow L_1^i$ and $\psi(L_2) \Rightarrow L_2^i$ and $\psi(L_3) \Rightarrow L_3^i$. Thus $\psi(L_1) \Rightarrow \psi(L_2) \vee \psi(L_3) \Rightarrow L_2^i \vee L_3^i$ and since $\wedge$ is the greatest lower bound, $\psi(L_1) \Rightarrow L_1^i \wedge (L_2^i \vee L_3^i) = L_1^{i+1}$ as needed.

□

2.3 Protocol Selection

The protocol selection phase of Viaduct assigns a protocol to each program component. Formally, a *protocol assignment* is a function $\Pi : (\mathbb{T} \cup \mathbb{X}) \rightarrow \mathbb{P}$ from temporaries and assignables to protocols. For a temporary t, $\Pi(t)$ is the protocol that executes the expression associated with t. Similarly, $\Pi(x)$ is the protocol that stores and responds to method calls on the data type instance bound to x.

$$\boxed{\Pi \models e : P} \qquad \boxed{\Pi \models s}$$

$$\frac{}{\Pi \models v : P} \qquad \frac{\mathrm{comm}(\Pi(t), P)}{\Pi \models t : P} \qquad \frac{\Pi \models a_i : P}{\Pi \models op_n(a_1, \ldots, a_n) : P}$$

$$\frac{\Pi \models a_i : \Pi(x)}{\Pi \models x.m(a_1, \ldots, a_n) : \Pi(x)} \qquad \frac{\Pi \models a : P}{\Pi \models \textbf{declassify } a \textbf{ to } \ell : P}$$

$$\frac{\Pi \models a : P}{\Pi \models \textbf{endorse } a \textbf{ from } \ell : P} \qquad \frac{}{\Pi \models \textbf{input}_\beta\ h : \mathrm{Local}(h)}$$

$$\frac{\Pi \models a : \mathrm{Local}(h)}{\Pi \models \textbf{output } a \textbf{ to } h : \mathrm{Local}(h)}$$

$$\frac{\begin{array}{c}\mathbb{L}(\Pi(t)) \Rightarrow \mathbb{L}(t) \\ \Pi \models e : \Pi(t) \qquad \Pi \models s\end{array}}{\Pi \models \textbf{let } t = e \textbf{ in } s} \qquad \frac{\begin{array}{c}\mathbb{L}(\Pi(x)) \Rightarrow \mathbb{L}(x) \\ \Pi \models a_i : \Pi(x) \qquad \Pi \models s\end{array}}{\Pi \models \textbf{new } x = D(a_1, \ldots, a_n) \textbf{ in } s}$$

$$\frac{\mathrm{hosts}(\Pi, s_1) \cup \mathrm{hosts}(\Pi, s_2) \subseteq \mathrm{visible}(\Pi(t)) \qquad \Pi \models s_1 \qquad \Pi \models s_2}{\Pi \models \textbf{if } t \textbf{ then } s_1 \textbf{ else } s_2}$$

$$\frac{\Pi \models s}{\Pi \models b : \textbf{loop } s} \qquad \frac{}{\Pi \models \textbf{break } b} \qquad \frac{\Pi \models s_1 \qquad \Pi \models s_2}{\Pi \models s_1; s_2} \qquad \frac{}{\Pi \models \textbf{skip}}$$

Figure 2.8: Rules for the validity of a protocol assignment.

2.3.1 Validity of Protocol Assignments

Figure 2.8 outlines the conditions under which a protocol assignment is valid. The judgement $\Pi \models e : P$ means that expression e can be executed by protocol P under assignment Π. Similarly, the judgement $\Pi \models s$ means that Π is a valid assignment for statement s.

We now describe the rules for validity. The rule for temporaries states that t can

$$\boxed{\Pi(s) : 2^{\mathbb{P}}} \qquad\qquad \boxed{\text{hosts}(\Pi, s) : 2^{\mathbb{H}}}$$

$$\begin{aligned}
\Pi(\textbf{let } t = e \textbf{ in } s) &= \Pi(t) \cup \Pi(s) \\
\Pi(\textbf{new } x = D(a_1, \ldots, a_n) \textbf{ in } s) &= \Pi(x) \cup \Pi(s) \\
\Pi(\textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2) &= \Pi(s_1) \cup \Pi(s_2) \\
\Pi(b : \textbf{loop } s) &= \Pi(s) \\
\Pi(\textbf{break } b) &= \Pi(b : \textbf{loop } s) \\
\Pi(s_1; s_2) &= \Pi(s_1) \cup \Pi(s_2) \\
\Pi(\textbf{skip}) &= \emptyset
\end{aligned}$$

$$\text{hosts}(\Pi, s) = \textstyle\bigcup_{P \in \Pi(s)} \text{hosts}(P)$$

Figure 2.9: Protocols and hosts involved in the execution of a statement. Here, $\text{hosts}(P)$ is the set of hosts that protocol P runs on, which is specified individually for each protocol.

$$\begin{aligned}
&\text{cost}(\Pi, \textbf{let } t = e \textbf{ in } s) = \\
&\quad \text{c}_{\text{exec}}(\Pi(t), e) + \sum_{P \in \text{readers}(\Pi, t, s)} \text{c}_{\text{comm}}(\Pi(t), P) + \text{cost}(\Pi, s) \\
&\text{cost}(\Pi, \textbf{if } a \textbf{ then } s_1 \textbf{ else } s_2) = \max(\text{cost}(\Pi, s_1), \text{cost}(\Pi, s_2)) \\
&\text{cost}(\Pi, b : \textbf{loop } s) = W_{\text{loop}} \times \text{cost}(\Pi, s) \\
&\text{cost}(\Pi, s_1; s_2) = \text{cost}(\Pi, s_1) + \text{cost}(\Pi, s_2) \\
&\text{cost}(\Pi, s) = 0 \text{ otherwise}
\end{aligned}$$

Figure 2.10: Abstract cost model.

$$\begin{aligned}
\text{visible}(\text{Local}(h)) &= \{h\} \\
\text{visible}(\text{Replicated}(H)) &= H \\
\text{visible}(\text{Commitment}(h_{\text{p}}, h_{\text{v}})) &= \{h_{\text{p}}\} \\
\text{visible}(\text{ZKP}(h_{\text{p}}, h_{\text{v}})) &= \{h_{\text{p}}\} \\
\text{visible}(\text{MAL-MPC}(H)) &= \emptyset \\
\text{visible}(\text{SH-MPC}(H)) &= \emptyset
\end{aligned}$$

Figure 2.11: For each protocol, the set of hosts for whom data stored in that protocol is visible.

only be read by protocol P if $\Pi(t)$, the protocol storing t, can communicate with P, written $\mathrm{comm}(\Pi(t), P)$. Not all pairs of protocols can communicate; the customizable *protocol composer*, discussed further in §2.4.1, defines the valid set of protocol compositions.

Other rules restrict where certain expressions can be executed. A method call on x must be executed by $\Pi(x)$, the protocol that stores x. Similarly, input/output expressions must be executed locally on the relevant host.

The rules for temporary and assignable declarations ensure that the protocol selected for a temporary or assignable has enough authority to securely store it. Formally, the label $\mathbb{L}(\Pi(t))$ of the protocol storing temporary t must act for ($\Rightarrow$) the minimum required authority label $\mathbb{L}(t)$ computed for t in §2.2.2 (and similarly for assignables). Labels $\mathbb{L}(\Pi(t))$ are the ones explained in Table 2.1.

The rule for conditional statements ensures that all hosts involved in the execution of a conditional statement (Figure 2.9) can learn which branch is taken. The side condition requires that the hosts participating in the execution of either branch (s_1 or s_2) must be contained in the set of hosts for whom the result computed by the protocol executing the guard is visible ($\mathrm{visible}(\Pi(t))$). This side condition is only checked if the guard is a temporary; if the guard is a literal, then the side condition is trivially true (any hosts can learn the value of a literal). Where necessary, the Viaduct compiler removes these guard visibility constraints by multiplexing [75] conditional statements into straight-line code. This allows, for example, the compilation of conditionals with secret guards that require execution in MPC.

Note that $\mathrm{visible}(P) \subseteq \mathrm{hosts}(P)$—that is, it can be the case that for some of the hosts involved in a protocol P, the result of some computation executed in the protocol

is not visible to them. Figure 2.11 shows the definition of $\mathrm{visible}(\cdot)$ for a variety of protocols. For example, if data is stored in a commitment or a ZKP protocol, then it is only visible to the prover. For data stored in MPC, the data is visible to *none* of the hosts.

2.3.2 Cost of Protocol Assignments

There can be many valid protocol assignments that securely realize a source program. To select an optimal assignment, Viaduct attributes a cost to each assignment using an abstract cost model, shown in Figure 2.10. Developers can instantiate the abstract model by modifying the customizable *cost estimator*, which specifies $\mathrm{c}_{\mathrm{exec}}(P, s)$, the cost of executing statement s in protocol P; $\mathrm{c}_{\mathrm{comm}}(P_1, P_2)$, the cost of communicating between P_1 and P_2; and the global constant W_{loop}, the number of times a loop is assumed to execute when its iteration count is not statically known.

Our implementation configures $\mathrm{c}_{\mathrm{exec}}$ to assign a small cost to executing "in the clear" and a large cost to the use of cryptography, so the compiler avoids the use of cryptography except when required for security. We also configure the communication cost $\mathrm{c}_{\mathrm{comm}}$ to minimize data movement. For example, a frequently accessed public variable would be replicated on two hosts so that each host has a local copy. Placing the variable only on one of the hosts could reduce storage cost but entails frequently sending its value to the other host.

$$
\begin{array}{lrcl}
\text{Constraint} & B & ::= & \text{true} \mid \text{false} \mid B \wedge B \mid B \vee B \mid \neg B \\
\text{Cost Expression} & C & ::= & n \mid C + C \mid n \times C \mid \max(C, C) \mid B \mathbin{?} C
\end{array}
$$

Figure 2.12: Abstract syntax for constraints and cost expressions in an optimization problem.

$$
\begin{aligned}
\varphi \models \alpha_{t,P} &\iff \varphi(\alpha_{t,P}) = \text{true} \\
\varphi \models \alpha_{x,P} &\iff \varphi(\alpha_{x,P}) = \text{true} \\
\varphi \models B_1 \wedge B_2 &\iff \varphi \models B_1 \text{ and } \varphi \models B_2 \\
\varphi \models B_1 \vee B_2 &\iff \varphi \models B_1 \text{ or } \varphi \models B_2 \\
\varphi \models \neg B &\iff \varphi \not\models B
\end{aligned}
$$

Figure 2.13: Tarski-style semantics for constraint satisfaction.

2.3.3 Computing an Optimal Protocol Assignment

To compute an optimal protocol assignment given a program s, the Viaduct compiler constructs a optimization problem over a set of assignment proposition variables $\alpha_{t,P}$, which when true witnesses the fact that temporary t is assigned to be executed at protocol P, and $\alpha_{x,P}$, which when true witnesses the fact that assignable x is stored at protocol P. The optimization problem consists of a conjunction of constraints B

$$
\begin{aligned}
[\![n]\!]_\varphi &= n \\
[\![C_1 + C_2]\!]_\varphi &= [\![C_1]\!]_\varphi + [\![C_2]\!]_\varphi \\
[\![n \times C]\!]_\varphi &= n \times [\![C]\!]_\varphi \\
[\![B \mathbin{?} C]\!]_\varphi &= \begin{cases} [\![C]\!]_\varphi & \text{if } \varphi \models B \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.14: Evaluation function for cost expressions.

$$\boxed{s \rightsquigarrow (B, C, PH)} \qquad \textbf{skip} \rightsquigarrow (\text{true}, 0, \bot) \qquad \textbf{break}\ b \rightsquigarrow (\text{true}, 0, \bot)$$

$$\frac{s_1 \rightsquigarrow (B_1, C_1, PH_1) \qquad s_2 \rightsquigarrow (B_2, C_2, PH_2)}{s_1; s_2 \rightsquigarrow (B_1 \wedge B_2, C_1 + C_2, PH_1 \sqcup PH_2)} \qquad \frac{s \rightsquigarrow (B, C, PH)}{b : \textbf{loop}\ s \rightsquigarrow (B, W_{\text{loop}} \times C, PH)}$$

$$\frac{\begin{array}{c} s_1 \rightsquigarrow (B_1, C_1, PH_1) \qquad s_2 \rightsquigarrow (B_2, C_2, PH_2) \\ B_{\text{v}} = \bigwedge_{h \in H} (PH_1 \sqcup PH_2)(h) \implies \text{oneof}(\{\alpha_{t,P} \mid P \in \text{viable}(t) \wedge h \in \text{visible}(P)\}) \end{array}}{\textbf{if}\ t\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow (B_{\text{v}} \wedge B_1 \wedge B_2, \max(C_1, C_2), PH_1 \sqcup PH_2)}$$

$$\frac{\begin{array}{c} s \rightsquigarrow (B_s, C_s, PH_s) \qquad VP = \text{viable}(t) \\ RS = \text{reads}(e) \qquad f_\alpha = \lambda P.\alpha_{t,P} \qquad B_{\text{exec}} = \text{constrain-exec}(VP, f_\alpha) \\ B_{\text{read}} = \text{constrain-read}(VP, f_\alpha, RS) \qquad B_{\text{method}} = \text{constrain-method}(t, e) \\ C = \text{possible-cost}(VP, f_\alpha, \lambda P.\, \text{c}_{\text{exec}}(P, e), RS) \qquad PH = \text{possible-hosts}(VP, f_\alpha) \end{array}}{\textbf{let}\ t = e\ \textbf{in}\ s \rightsquigarrow (B_{\text{exec}} \wedge B_{\text{read}} \wedge B_{\text{method}} \wedge B_s, C + C_s, PH \sqcup PH_s)}$$

$$\frac{\begin{array}{c} s \rightsquigarrow (B_s, C_s, PH_s) \\ VP = \text{viable}(x) \qquad RS = \text{reads}(\{a_1, \ldots, a_n\}) \qquad f_\alpha = \lambda P.\alpha_{x,P} \\ B_{\text{exec}} = \text{constrain-exec}(VP, f_\alpha) \qquad B_{\text{read}} = \text{constrain-read}(VP, f_\alpha, RS) \\ C = \text{possible-cost}(VP, f_\alpha, \lambda P.\, \text{c}_{\text{exec}}(P, D), RS) \qquad PH = \text{possible-hosts}(VP, f_\alpha) \end{array}}{\textbf{new}\ x = D(a_1, \ldots, a_n)\ \textbf{in}\ s \rightsquigarrow (B_{\text{exec}} \wedge B_{\text{read}} \wedge B_s, C + C_s, PH \sqcup PH_s)}$$

Figure 2.15: Rules to generate optimization problem.

and cost expression C. The syntax for these are given in Figure 2.12. Constraints are Boolean expressions over the set of assignment variables; cost expressions describe "linear" arithmetic expressions. Additionally, the conditional cost expression $B\ ?\ C$ has cost denoted by C when B is satisfied; otherwise it has 0 cost.

A solution to the problem is a model φ that maps assignment variables to truth values such that B is satisfied. Satisfaction is captured by judgment $\varphi \models B$ defined in Figure 2.13. When constraint B and cost expression C are generated from a statement s, then a solution φ corresponds to a protocol assignment Π such that

$$\Pi(t) = P \iff \varphi \models \alpha_{t,P} \qquad \Pi(x) = P \iff \varphi \models \alpha_{x,P} \qquad \Pi \models s.$$

Note that constraints generated from s ensure that Π is well-defined. That is, if $\varphi \models \alpha_{t,P}$ then for any $P' \neq P$, $\varphi \not\models \alpha_{t,P}$; a similar constraint holds for when $\varphi \models \alpha_{x,P}$.

An *optimal* solution φ_{opt} to the optimization problem is one that minimizes the value of the cost expression as given by an evaluation function $[\![\cdot]\!]_{\varphi}$ that maps cost expressions to $\mathbb{N}$-valued cost valuations, as given in Figure 2.14:

$$\varphi_{\text{opt}} = \underset{\varphi \in \{\varphi \mid \varphi \models B\}}{\arg\min} \; [\![C]\!]_{\varphi}\,.$$

In the prototype implementation of the Viaduct compiler (§2.5), we use an off-the-shelf solver to compute φ_{opt} given a constraint B and cost expression C.

Finally, from an optimal solution we can construct an *optimal* protocol assignment Π_{opt} such that

$$\Pi_{\text{opt}} = \underset{\Pi \in \{\Pi \mid \Pi \models s\}}{\arg\min} \; \text{cost}(\Pi, s).$$

Protocol Factory. To construct the optimization problem, the compiler draws the set of available protocols from the customizable *protocol factory*. Developers wishing to add new protocols to Viaduct must extend the protocol factory so that the compiler can generate assignments with these protocols during protocol selection.

The protocol factory defines a function $\text{viable} : \mathbb{T} \cup \mathbb{X} \to 2^{\mathbb{P}}$ that returns a set of viable protocols that can execute a let-binding or declaration. Though customizable, the set of viable protocols must still respect the authority requirements of the computed labels for temporaries and assignables; that is, it must be the case that

$$\forall P \in \text{viable}(t).\mathbb{L}(P) \Rightarrow \mathbb{L}(t) \quad \text{and} \quad \forall P \in \text{viable}(x).\mathbb{L}(P) \Rightarrow \mathbb{L}(x).$$

The protocol factory allows developers to specify limitations regarding the use of particular protocols, which is important because in general cryptographic protocols are

limited in the storage and computation they can perform. For example, commitment protocols may be unable to compute over commitments. Other protocols may lack support for certain operators.

Constructing the Optimization Problem. The compiler uses set of syntax-directed rules to generate an optimization problem from a program. The rules have the form $s \leadsto (B, C, PH)$, which means that statement s generates constraint B, cost expression C, and participating hosts map PH, which maps each host to a proposition that determines the conditions when the host is deemed to be participating in the execution of s. The distinguished map $\bot$ maps every host to false, meaning that no hosts are participating under any circumstance. Maps can be merged using the $\sqcup$ operator, which returns a map where a host is deemed to be participating when either input map deem it to be participating:

$$(PH_1 \sqcup PH_2)(h) = PH_1(h) \vee PH_2(h).$$

Before we discuss the constraint generation rules, we define some auxilliary functions. We define a function oneof$(\cdot)$ that takes as input a set of constraints and then returns a new constraint that ensures exactly one of the input constraints is true:

$$\text{oneof}(\{B_1, \ldots, B_n\}) = \bigvee_{i=1}^{n} \left(\bigwedge_{j \neq i} (\neg B_j) \wedge B_i \right).$$

The function $\text{can-send}(P)$ returns the set of protocols that can send data to a protocol P:

$$\text{can-send}(P) = \{P' \mid \text{comm}(P', P)\}.$$

We lift the function viable to consider only the set of viable functions that can send data to some protocol:

$$\text{viable}(t, P) = \text{viable}(t) \cap \text{can-send}(P) \qquad \text{viable}(x, P) = \text{viable}(x) \cap \text{can-send}(P).$$

Finally, we lift the function reads from single expressions to a set of expressions:

$$\text{reads}(\{e_1, \ldots, e_n\}) = \bigcup_{i=1}^{n} \text{reads}(e_i).$$

Figure 2.15 shows the rules for constructing the optimization problem. The rule for **skip** generates a constraint true, cost 0, and participating hosts map $\bot$; this means that it does not generate any constraints about the protocol assignment, does not cost anything to execute, and no hosts execute it. The rule for sequencing statement $s_1; s_2$ requires that the constraints generated for s_1 and s_2 both hold; combines their costs together; and generates a hosts map such that a host participates in executing the sequence if it participates in executing either s_1 or s_2.

Meanwhile, the rule for conditionals requires that the constraints for both branches s_1 and s_2 must hold; the conditional accrues the cost of the most expensive branch; and also generates a hosts map such that a host participates in executing the conditional if it participates in executing either branch. Furthermore, rule generates a constraint for the visibility of the guard by ensuring that if a host h participates in executing either branch, then the guard must be computed by a viable protocol P whose result is visible to the host (i.e., $h \in \text{visible}(P)$).

The rules for let-bindings and assignable declarations are similar and rely on a few auxilliary functions to define their generated constraints, cost expressions, and participating hosts maps. First, the constrain-exec function takes in a set of viable protocols and a function f_α that takes in a viable protocol and returns its associated assignment variable. In turn constrain-exec returns a constraint that ensures that exactly one of the assignment variables associated with a viable protocol is true.

$$\text{constrain-exec}(VP, f_\alpha) = \text{oneof}(\{f_\alpha(P) \mid P \in VP\}).$$

Next, the constrain-read function takes in a set of viable protocols VP, a function

f_α that takes in a viable protocol and returns its associated assignment variable, and a set of read temporaries RS. In turn, constrain-read returns a constraint that ensures that for every viable protocol P, the let-bindings of the temporary variables in RS must be assigned to a protocol that can send data to P.

$$\text{constrain-read}(VP, f_\alpha, RS) = \bigwedge_{P \in VP} \left(f_a(P) \implies \bigwedge_{t' \in RS} \text{oneof}(\{\alpha_{t',P'} \mid P' \in \text{viable}(t', P)\}) \right).$$

Next, the constrain-method function ensures that a method call whose result is bound to temporary t is executed by the same protocol storing the receiver of the call:

$$\text{constrain-method}(t, e) = \begin{cases} \bigwedge_{P \in \text{viable}(x)} \alpha_{t,P} \iff \alpha_{x,P} & \text{if } e = x.m(a_1, \ldots, a_n) \\ \text{true} & \text{otherwise} \end{cases}$$

Next, the possible-cost function takes in a set of viable protcols VP; a function f_α that takes in a viable protocol and returns and assignment variable; a function f_c that takes in a protocol P and returns the execution cost associated with P; and a set of read temporaries RS. In turn, it returns a cost expression that sets the execution and communication cost of choosing from the set of viable protocols in VP and the set of viable protocols for each read temporary in RS.

$$\text{possible-cost}(VP, f_\alpha, f_c, RS) = \sum_{P \in VP} f_\alpha(P) \ ? \ \left(f_c(P) + \sum_{t' \in RS} \sum_{P' \in \text{viable}(t', P)} \alpha_{t',P'} \ ? \ \text{c}_{\text{comm}}(\Pi(t), P') \right).$$

Finally, the possible-hosts function takes in a set of viable protocols VP and a function f_α that takes in a viable protocol and returns its associated assignment variable. In turn, the function returns a participating hosts map where each host is mapped to

the disjunction of assignment variables of viable protocols where the host participates in the execution of the protocol.

$$\text{possible-hosts}(VP, f_\alpha) = \bigsqcup_{P \in VP} \bigsqcup_{h \in \text{hosts}(P)} \bot[h \mapsto f_\alpha(P)].$$

Example. Consider the following source program to be executed by hosts a and b:

$$\textbf{let } t_1 = 1 + 1 \textbf{ in let } t_2 = t_1 \times 2 \textbf{ in skip}$$

and the following data from the compiler's extension points:

1. $\text{viable}(t_1) = \{P_1, P_3\}$, $\text{viable}(t_2) = \{P_1, P_2\}$
2. $\text{hosts}(P_1) = \{a\}$, $\text{hosts}(P_2) = \{b\}$, $\text{hosts}(P_3) = \{a, b\}$
3. $\text{c}_{\text{exec}}(P_1, \_) = 5$, $\text{c}_{\text{exec}}(P_2, \_) = 5$, $\text{c}_{\text{exec}}(P_3, \_) = 3$
4. $\text{c}_{\text{comm}}(P_1, P_1) = 0$, $\text{c}_{\text{comm}}(P_3, P_2) = 1$
5. $\text{comm}(P_1, P_1)$, $\neg\, \text{comm}(P_3, P_1)$
6. $\text{comm}(P_3, P_2)$, $\neg\, \text{comm}(P_1, P_2)$

Then the compiler generates the following constraint:

$$\begin{aligned}&\text{oneof}(\{\alpha_{t_1,P_1}, \alpha_{t_1,P_3}\}) \wedge \text{oneof}(\{\alpha_{t_2,P_1}, \alpha_{t_2,P_2}\}) \\ &\wedge (\alpha_{t_2,P_1} \implies \text{oneof}(\{\alpha_{t_1,P_1}\})) \wedge (\alpha_{t_2,P_2} \implies \text{oneof}(\{\alpha_{t_1,P_3}\}))\end{aligned}$$

and the following cost expression:

$$\begin{aligned}&\alpha_{t_1,P_1} \ ?\ \text{c}_{\text{exec}}(P_1, 1+1) + \alpha_{t_1,P_3} \ ?\ \text{c}_{\text{exec}}(P_3, 1+1) \\ &+ \alpha_{t_2,P_1} \ ?\ (\text{c}_{\text{exec}}(P_1, t_1 \times 2) + \alpha_{t_1,P_1} \ ?\ \text{c}_{\text{comm}}(P_1, P_1)) \\ &+ \alpha_{t_2,P_2} \ ?\ (\text{c}_{\text{exec}}(P_2, t_1 \times 2) + \alpha_{t_1,P_3} \ ?\ \text{c}_{\text{comm}}(P_3, P_2))\,.\end{aligned}$$

From this optimization problem the compiler then computes the optimal assignment Π_{opt} where $\Pi_{\text{opt}}(t_1) = P_3$ and $\Pi_{\text{opt}}(t_2) = P_2$. This assignment gives makes the cost expression evaluate to 9; the other valid protocol assignment Π would be such that $\Pi(t_1) = P_1$ and $\Pi(t_2) = P_1$, which makes the cost expression evaluate to 10.

2.4 Viaduct Runtime

Once it has computed a protocol assignment, the Viaduct compiler outputs a program where every let-binding and assignable declaration is annotated with the protocol that will execute it. This annotated program can be executed by the Viaduct runtime, which consists of an extensible interpreter that interacts with a set of *protocol back ends*, each of which implement a set of protocols. The interface for protocol back ends is straightforward: back ends must implement methods to execute let-bindings and assignable declarations, and methods to communicate with other protocol back ends.

Each host runs a copy of the interpreter with the annotated program as input. For each statement, the interpreter checks whether the host participates in its execution, as defined by $\text{hosts}(\Pi, \cdot)$—if not, the statement is treated like **skip**. If a host participates in executing a let-binding or a declaration, the interpreter calls the back end for the protocol assigned to the statement. To execute a conditional, the host retrieves the cleartext value of the guard from the protocol back end that stores it, and executes the appropriate branch. The validity rules for protocol assignments ensure the host is allowed to see the cleartext value, and that it is able to retrieve it.

2.4.1 Protocol Composition

The protocol back end executing a let-binding must send the computed value to back ends executing statements that read the bound temporary. How one back end sends a value to another depends on the protocols involved. For example, a statement executed in $\mathrm{Replicated}(h_1, h_2)$ reading a temporary computed in $\text{SH-MPC}(h_1, h_2)$ corresponds to executing an MPC circuit and revealing the output to the hosts. On the other hand, a temporary computed in $\mathrm{Local}(h_3)$ might not meaningfully be read by a statement executed under $\text{SH-MPC}(h_1, h_2)$ as it is unclear how the MPC back end should read local data from an unrelated host.

Viaduct uses the customizable *protocol composer* to define the set of source and destination protocols that can communicate. The composer translates communication between two protocols to a set of messages between hosts participating in the protocols. Developers who want to extend Viaduct with support for a new protocol must enumerate the set of allowed compositions for the protocol and ensure that such compositions are secure.

Formally, the protocol composer translates communication between two protocols P_1 and P_2 to a set of messages, each of the form $(P_1, h_1) \xrightarrow{a} (P_2, h_2)$, where the back end for protocol P_1 at host h_1 sends a message to the back end for protocol P_2 at host h_2 along port a. For a pair (P, h), it must be the case that $h \in \mathrm{hosts}(P)$. The Viaduct runtime handles the delivery of these messages between back ends.

Each protocol provides a set of ports that define how its back end processes input from another protocol back end. The ZKP protocol, for instance, has two ports: a secret input port, and a public input port. The ZKP back end treats data from the secret input port as the secret input of the prover, while it treats data from its public input port as

data known to both the prover and verifier.

Recalling the previous example, when $\text{SH-MPC}(h_1, h_2)$ sends a value to $\text{Replicated}(h_1, h_2)$, the MPC back ends in h_1 and h_2 jointly execute a circuit in an MPC protocol. The MPC back end at h_1 then sends the revealed circuit output to the cleartext back end (which implements the Replicated protocol) at h_1 along its cleartext port. There is a corresponding message between the MPC and cleartext back ends at h_2. Step (3) in Figure 2.4, which depicts execution of the historical millionaires' problem, shows this protocol composition in the context of a larger program.

We know give some examples of protocol composition, where the sending protocol is denoted as s and the receiving protocol is denoted as r. The ct port of various protocols stands for cleartext input; the in port of the MPC protocol represents secret input from a host; the cc port of the Commitment protocol represents creating a commitment; the occ and ohc ports of the Local protocol respectively represent receiving the cleartext value of an opened commitment and the commitment itself.

- $s = \text{Local}(h_1), r = \text{SH-MPC}(h_1, h_2)$. The induced communication is $(s, h_1) \xrightarrow{in} (r, h_1)$. Intuitively, this means that an input gate for h_1 in created the current MPC circuit.
- $s = \text{Local}(h_\text{p}), r = \text{Commitment}(h_\text{p}, h_\text{v})$. The induced communication is $(s, h_\text{p}) \xrightarrow{cc} (r, h_\text{p})$. Intuitively, this means that h_p creates and then sends a commitment to h_v.
- $s = \text{Replicated}(h_1, h_2), r = \text{Local}(h_1)$. The induced communication is $(s, h_1) \xrightarrow{ct} (r, h_1)$. Intuitively, this means that h_1 locally reads data that is replicated between h_1 and h_2.
- $s = \text{SH-MPC}(h_1, h_2), r = \text{Replicated}(h_1, h_2)$. The induced communication is

$(s, h_1) \xrightarrow{ct} (r, h_1)$ and $(s, h_2) \xrightarrow{ct} (r, h_2)$. Intuitively, this means that the current MPC circuit is executed and its result is sent to both h_1 and h_2.

- $s = \text{Commitment}(h_\text{p}, h_\text{v}), r = \text{Local}(h_\text{v})$. The induced communication is $(s, h_\text{p}) \xrightarrow{occ} (r, h_\text{v})$ and $(s, h_\text{v}) \xrightarrow{ohc} (r, h_\text{v})$. Intuitively, this means that the commitment created by h_p and stored by h_v is opened.
- $s = \text{ZKP}(h_\text{p}, h_\text{v}), r = \text{Local}(h_\text{v})$. The induced communication is $(s, h_\text{v}) \xrightarrow{ct} (r, h_\text{v})$. Intuitively, this means that the result of a computation performed by h_p and its accompanying zero-knowledge proof is sent to h_v, who then uses the proof to validate the result.

These examples illustrate our insight that protocol composition is a general abstraction to represent the use of cryptographic mechanisms. The creation of a commitment and its opening; the execution of an MPC circuit and the revealing of its output; a prover sending a zero-knowledge proof to a verifier—all of these are captured by a composition of one protocol with another.

2.5 Implementation

We implemented the Viaduct compiler in about 20 KLoC of Kotlin code, which includes code for the parser, the label constraint solver, protocol selection, and the runtime system. The code written against the compiler's extension points—the protocol factory, the protocol composer, the cost estimator, and the protocol back ends—runs to about 4 KLoC. Viaduct uses the Z3 SMT solver [35] to solve the optimization problem generated during protocol selection.

The compiler supports the more liberal surface syntax seen in Figure 2.2 and Fig-

ure 2.3, as well as functions with bounded polymorphism on parameter labels. The compiler specializes functions at each call site, allowing different compiled implementations for the same function.

We implemented four protocol back ends for Viaduct:

Local/Replicated. The cleartext back end executes code in Local and Replicated protocols. It maintains a store for objects that directly represent the temporaries and assignables of the source program. Computations performed by the cleartext back end are executed directly.

SH-MPC. This back end links Viaduct to ABY, a library for two-party semi-honest MPC [37]. It maintains a store of gate objects that represent circuit components to be executed by ABY. Computations performed by the back end build gate objects that represent the operation performed (e.g., an addition in the source program creates an ADD gate).

The ABY back end executes a circuit when the result of some gate object computed at MPC must be communicated to another protocol, as specified by the protocol composer extension point (e.g. MPC sends the result of a computation to the Replication protocol). Given a gate object set as the output of a circuit, the back end computes the (transitive) dependencies of the gate, all the way back to input gates (created when a protocol—e.g. Local—sends data to the *in* gate of the MPC protocol). The back end then executes the dependencies until the output gate can be executed. To avoid recomputation of shared subcircuits, the back end partitions the circuit into subcircuits and executes these in topological order (where edges between subcircuits are induced when the result of a subcircuit is used as input by another subcircuit).

The ABY framework supports execution of circuits in three different schemes—arithmetic sharing, boolean sharing, and Yao's garbled circuits—as well as conversions between these, allowing for execution of mixed-protocol circuits. Viaduct represents each scheme as a separate protocol, but all three are implemented by a single back end. To generate efficient mixed circuits, we follow Demmler et al. [37] and Ishaq et al. [57] and estimate inputs to the cost estimator by measuring execution time of individual operations under a particular scheme and conversions between schemes. We perform measurements for two settings: low-latency, high-bandwidth (LAN), and high-latency, low-bandwidth (WAN).[2] Thus the cost estimator has two modes, each of which optimizes compiled programs for a specific network environment.

Commitment. This back end manages commitments, implemented using SHA-256 hashes of data along with a nonce. The back end for the commitment creator maintains a store of cleartext values along with metadata for commitments. The back end for the commitment receiver maintains the set of commitments, as hashes. The commitment back end cannot support computation.

ZKP. This back end links to libsnark [1], a library for zkSNARKs (zero-knowledge Succinct Non-interactive ARguments of Knowledge). This back end maintains a store of circuit gate objects. The prover and verifier both manage cleartext values for the public inputs to the proof, while only the prover manages cleartext values for the secret inputs. To ensure the prover cannot modify secret inputs mid-execution, all secret inputs are "committed" by sending their hash to the verifier. All proofs that use a secret input then include a clause that equates the input to the pre-image of the hash

[2] Existing work such as Büscher et al. [18] and Ishaq et al. [57] focus on optimizing mixed circuits for ABY specifically, and as such these employ more sophisticated reasoning about cost for ABY circuits. We consider it future work to incorporate such techniques into Viaduct.

	Protocols			Selection	
Benchmark	**LAN / WAN**	**LoC**	**Ann**	**Vars**	**Time**
battleship	RZ / RZ	79	12	1022	1.0
bet	CLRY / CLRY 79	7	1022	1.0	
biometric match	ALRY / ALRY	40	8	708	2.0
guessing game	RZ / RZ	16	6	193	0.4
HHI score	ALRY / LRY	22	3	285	1.1
historical millionaires	LRY / LRY	17	3	187	0.7
interval	RYZ / RYZ	45	9	660	2.8
k-means	ARY / RY	82	3	1684	7.9
k-means (unrolled)	ARY / RY	174	3	3629	29.0
median	RY / RY	36	6	386	1.0
rock-paper-scissors	CR / CR	56	6	741	1.0
two-round bidding	LRY / LRY	34	4	575	1.7

Table 2.3: Benchmark programs. **Protocols** give the protocols used in the compiled program for either the LAN or WAN setting. Legend for protocols used: **A, B, Y**–ABY arithmetic/boolean/Yao sharing; **C**–Commitment; **L**–Local; **R**–Replicated; **Z**–ZKP. **Ann** gives the minimum number of label annotations needed to write the program. **Selection** gives the number of symbolic variables and run time in seconds for protocol selection, averaged across five runs.

held by the verifier.

The libsnark library requires proving and verifying keys to be generated for each unique circuit before the protocol is executed. The current prototype requires a "dummy" run of the compiled program to generate these keys.

2.6 Evaluation

To evaluate Viaduct, we address these research questions:

- Is the Viaduct source language expressive enough?

	Bool			Yao		
Benchmark	**LAN**	**WAN**	**Comm**	**LAN**	**WAN**	**Comm**
bio. match	3.6	95.9	56.0	2.8	7.1	52.3
HHI score	0.8	9.7	7.0	0.5	1.6	2.7
hist. million.	1.0	90.6	4.8	0.6	1.6	3.1
k-means	56.5	696.1	1273.1	44.4	117.4	1051.3
median	11.5	1098.7	197.1	12.8	35.4	327.8
2-R bidding	17.3	184.7	233.0	17.8	184.5	233.0

	Opt-LAN			Opt-WAN		
Benchmark	**LAN**	**WAN**	**Comm**	**LAN**	**WAN**	**Comm**
bio. match	**1.0**	**2.2**	**3.9**	same as Opt-LAN		
HHI score	**0.3**	1.1	**0.5**	**0.3**	**0.9**	0.6
hist. million.	**0.3**	**0.7**	**0.005**	same as Opt-LAN		
k-means	**17.7**	**35.8**	**180.0**	same as Yao		
median	**0.7**	**31.7**	**1.0**	same as Opt-LAN		
2-R bidding	**3.1**	**155.5**	**4.7**	same as Opt-LAN		

Table 2.4: Run time (in seconds) and communication (in MB) of select benchmark programs, averaged across five runs. **Bool** and **Yao** are naive assignments using boolean sharing and Yao sharing respectively to execute MPC computations. **Opt-LAN** and **Opt-WAN** are optimal assignments generated by Viaduct for the LAN and WAN setting respectively. Optimal time and communication for a benchmark and execution setting pair are in **bold**.

	LAN		WAN	
Benchmark	**Time**	**Slowdown**	**Time**	**Slowdown**
bio. match	0.4	150%	1.5	50%
HHI score	0.3	0%	1.0	10%
hist. million.	0.3	0%	0.7	0%
k-means	1.2	1380%	4.1	770%
median	0.5	40%	31.5	0%
2-R bidding	1.6	90%	154.7	0%

Table 2.5: Run time (in seconds) of LAN-optimized benchmarks hand-written to use ABY directly and the slowdown of running the same benchmarks through the Viaduct runtime in LAN and WAN settings.

- Is its compilation performance acceptable?
- Does it generate efficient distributed programs?
- How much does label inference reduce the annotation burden for programmers?
- What is the overhead of the runtime system?

Experiments used Dell OptiPlex 7050 machines with an 8-core Intel Core i7 7th Gen CPU and 16GB of RAM. Note that for experiments involving time measurements, the numbers reported are over 5 trials and the relative standard error is at most 6% of the sample mean.

2.6.1 Expressiveness of Source Language

Table 2.3 shows the benchmarks used for the experiments and the cryptography synthesized by Viaduct for each benchmark. Several are from prior work, rewritten in the Viaduct source language. Host configurations are either semi-honest, as in Figure 2.2, where hosts A and B trust each other for integrity; mutually distrusting as in Figure 2.3; or are "hybrid" configurations where A and B trust each other but host C is trusted by neither.

The benchmarks are as follows.

- **battleship**: model of the board game.
- **bet**: C bets who wins historical millionaires game between A and B.
- **biometric match**: compute the minimum distance of a sample to a database of biometric data.
- **guessing game**: same as in fig. 2.3.

- **HHI score**: computes Herfindahl–Hirschman index for market concentration (from [105]).
- **historical millionaires**: same as in fig. 2.2.
- **interval**: A and B compute interval of their combined points, then C attests its point is in the interval.
- **k-means**: cluster secret points from A and B.
- **k-means**: k-means with 3 unrolled iterations.
- **median**: compute median of A and B's lists (from [60]).
- **rock-paper-scissors**: A and B commit to moves and then play rock-paper-scissors for a fixed number of rounds.
- **two-round bidding**: A and B bid for a list of items.

Our benchmarks show that Viaduct can compile programs whose security demands a variety of cryptographic mechanisms. With hybrid configurations (interval, bet), Viaduct combines MPC and ZKP to implement different components of a single distributed program. Code for selected benchmarks can be found in Appendix A.

2.6.2 Scalability of Compilation

The two main phases of the Viaduct compiler are label inference and protocol selection. Our benchmarks indicate that the overhead of label inference is negligible: at most several hundred milliseconds. As seen in Table 2.3, the overhead for protocol selection is more significant, but still on the order of several seconds for most benchmarks. The longest running benchmark, k-means, performs most of its computations in MPC. In

this case, it may be harder to converge to the optimal solution since the solver generates a large mixed circuit, choosing between the three MPC schemes supported by ABY.

2.6.3 Cost of Compiled Programs

To show that Viaduct can compile efficient distributed programs, we chose a subset of our benchmarks requiring the use of MPC and compared the execution of optimal programs generated by Viaduct—for each benchmark, one optimized for local area networks (LAN) and another for wide area networks (WAN) —with naive protocol assignments that perform all computation in MPC. The naive ABY assignments use either boolean sharing or Yao garbled circuits, since arithmetic sharing can only perform arithmetic operations. We measured executions in a 1 Gbps LAN and simulated WAN (100 Mbps bandwidth and 50 ms latency). We configured ABY to use 32-bit integers and set its security parameter to 128 bits.

Table 2.4 summarizes our results. For some benchmarks (HHI score, hist. millionaires, median, two-round bidding), computation can be securely moved from MPC to cleartext protocols, making execution much more efficient. Even for benchmarks that require computations to be almost entirely in MPC (bio. match, k-means), Viaduct chooses efficient mixed circuits that perform much better than the naive assignments entirely in boolean sharing or Yao circuits. Viaduct replicates the result in Büscher et al. [18] (which specifically targets the ABY framework) in choosing a mix of arithmetic and Yao circuits as optimal assignments for the two benchmarks from that paper, with the exception of the k-means benchmark in the WAN setting.

2.6.4 Annotation Burden of Security Labels

Security-typed languages add some annotation burden when writing programs. In practice, labels on host declarations and downgrading operations suffice to specify intended security policies in Viaduct programs. To substantiate this claim, we created two versions of each benchmark program. In one, every variable has a label annotation; in the other, "erased" version, all such labels are omitted.

For all benchmarks, Viaduct generates the same compiled program for the fully labeled and the erased versions. Although the inferred labels for the erased programs are not exactly the same as in their manually labeled counterparts, the differences do not affect the protocols chosen.[3] The **Ann** column in Table 2.3 counts label annotations on erased programs. This is the minimum number of annotations needed to write the program: effectively, the number of downgrades plus the number of host declarations, each of which need an authority label. The table shows that the annotation burden is low: most benchmarks need only a few label annotations.

2.6.5 Overhead of Runtime System

The Viaduct runtime introduces some overhead compared to using cryptographic libraries like ABY directly. To measure this overhead, we translated Viaduct's LAN-optimized outputs for the MPC benchmarks in Table 2.4 to directly use the ABY framework's API. We then measured the performance of these hand-written programs in the

[3] This mostly occurs with data publicly known to hosts (e.g. loop indices, array lengths). Given hosts Alice and Bob, a fully-annotated benchmark might have label $A \sqcap B$ for the data, but Viaduct infers label $(A \wedge B)^{\leftarrow}$ in the erased version.

LAN and WAN settings.[4]

Table 2.5 gives running times for the hand-written programs and the overhead of using the Viaduct runtime. For most benchmarks, the Viaduct runtime incurs an overhead of at most 150% in the LAN setting; the overhead is reduced to at most 50% in the WAN setting where network delay is a more significant factor. This overhead is due to the cost of interpretation and dynamic circuit generation, and can be eliminated by moving circuit generation to compile time [72, 18].

The markedly larger overhead of the k-means benchmark is due to Viaduct recomputing intermediate results. The benchmark has 8 outputs; while Viaduct evaluates 8 smaller MPC circuits each with one output, the hand-written version evaluates one larger circuit with 8 outputs, taking advantage of shared intermediate computations. The compiler could, with additional analysis, determine when output gates can be grouped and executed in the same circuit. We leave this to future work.

2.7 Related Work

Compilation to Cryptographic Protocols. The idea of compiling a high-level program to a cryptographic protocol has been explored in the context of multiparty computation [55] (e.g., Fairplay [75], SCVM [71], ObliVM [72], OblivC [110], Wysteria [89], HyCC [18], SCALE-MAMBA [5]), and that of zero-knowledge proofs (e.g., Pinocchio [85], Geppetto [30], Buffet [106], xjSNARK [64]). Earlier work is generally limited to the domain of a particular fixed cryptographic task (e.g., MPC or ZKP); Viaduct's novelty is synthesizing efficient protocols *across* cryptographic tasks. Like SCVM [71],

[4]Running LAN-optimized programs in the WAN setting does not skew the results since Table 2.4 shows that LAN-optimized programs perform roughly the same as WAN-optimized programs in the WAN setting.

Viaduct can synthesize "hybrid" programs that perform computations locally, replicated between hosts, or under MPC. This is impossible in the simple two-point label model that many MPC compilers [5, 72] use, which only distinguish between public (low) and secret (high) information. Viaduct also does not fix the number of hosts in a program (unlike [72, 71, 75]), nor fix compiling programs only under a semi-honest or malicious setting (unlike [89, 72, 71, 64, 106, 85]).

Program Partitioning. Another line of related work [112, 114, 42, 43] describes distributed computations using sequential programs and captures security requirements using information-flow labels. The Jif/split compiler [112, 114] synthesizes simple cryptographic primitives such as cryptographic commitments to satisfy security constraints that would otherwise be impossible without relying on trusted principals. Unlike Viaduct, Jif/split is not extensible to new protocols. Later work [42, 43] proves computational soundness for a similar system under a strong attacker that controls the network and some of the hosts. However, this work does not support replicating computations (only *data* replication is supported), or the other protocols that Viaduct supports.

2.8 Summary

The prototype implementation of the Viaduct compiler compiles high-level, security-typed programs into efficient distributed programs that employ a variety cryptographic mechanisms to ensure security. With the unified abstraction of information flow labels, the compiler is *extensible* and can support multiple cryptographic mechanisms, a first for compilers that target secure computation. Our evaluation shows that the approach is practical, and paves the way to a unified framework for developing distributed pro-

grams with strong security requirements.

CHAPTER 3

VIADUCT-HE: A COMPILER FROM ARRAY PROGRAMS TO VECTORIZED HOMOMORPHIC ENCRYPTION

Homomorphic encryption (HE), which allows computations to be performed on encrypted data, has recently emerged as a viable way for securely offload computation. Efficient libraries [25] and hardware acceleration [90, 92] have improved performance to be acceptable for practical use in a diverse range of applications such as the Password Monitor in the Microsoft Edge web browser [66], privacy-preserving machine learning [47], privacy-preserving genomics [63], and private information retrieval [77].

Writing programs to be executed under HE, however, remains a forbidding challenge [103]. In particular, modern HE schemes support data encodings that allow for single-instruction, multiple data (SIMD) computation with very long vector widths but limited data movement capability.[1] SIMD parallelism allows developers to recoup the performance loss of executing programs in HE, but taking advantage of this capability requires significant expertise: efficient vectorized HE programs requires carefully laying out data in ciphertexts and interleaving data movement operations with computations. There is a large literature on efficient, expert-written vectorized HE implementations [58, 47, 17, 3, 62].

Prior work has developed compilers to ease the programmability burden of HE, but most work has targeted specific applications [33, 101, 12, 11, 3, 73], or focuses on challenges other than vectorization [34, 68, 32, 6]. Some HE compilers do attempt to generate vectorized implementations for arbitrary programs, but either fix simple data layouts for all programs [104] or require users to provide at least *some* information about complex data layouts [31, 74].

[1] Also known as ciphertext "packing" or "batching" in the literature.

We make the important observation that the complex, expert-written data layouts targeting specific applications in prior work are made possible by *array-level* reasoning. That is, given an array as input to an HE program, searching for an efficient layout amounts to asking such questions as "should this dimension of the array be vectorized in a single ciphertext, or be exploded along multiple ciphertexts?" This kind of reasoning is not reflected in the prior work on HE compilers, but as we show, it enables vectorized HE implementations with expert-level efficiency.

With this in mind, we reframe the problem of compiling efficient vectorized HE programs as two separate problems. First, a program must be "tensorized" and expressed as an *array program*. In many cases, this step is actually unnecessary, since the program can already be naturally expressed as operations over arrays. This is true for many HE applications, such as secure neural network inference. Once expressed as a computation over arrays, the space of possible vectorization schedules for the program can be given a simple, well-defined representation. This makes the "last-mile" vectorization of array programs much more tractable than the vectorization of arbitrary imperative programs.

This "tensorize-then-vectorize" approach is arguably already present in the literature. For example, Malik et al. [73] developed an efficient vectorized HE implementation for evaluating decision forests by expressing the evaluation algorithm as a sequence of element-wise array operations and matrix–vector multiplication, and then using an existing kernel [53] to implement matrix–vector multiplication efficiently.

In this paper, we aim to tackle the challenge of generating vectorized HE implementations for array programs. To this end, we propose Viaduct-HE, a vectorizing HE compiler for an array-oriented source language. Viaduct-HE simultaneously generates the complex data layouts and operations required for efficient HE implementations.

Unlike in prior work [31, 74], this process is completely automatic: the compiler needs no user hints to generate complex layouts. The compiler leverages the high-level array structure of source programs to give a simple representation for possible vectorization schedules, allowing it to efficiently explore the space of schedules and find efficient data layouts. Once a schedule has been found, the compiler can further optimize the program by translating it to an intermediate representation amenable to term rewriting.

Viaduct-HE is designed to be extensible: after optimization, the compiler translates circuits into a loop nest representation designed for easy translation into operations exposed by HE libraries, allowing for the straightforward development of back ends that target new HE implementations. The compiler also has well-defined extension points for customizing the exploration of vectorization schedules and for estimating the cost of HE programs.

3.1 Background on Homomorphic Encryption

Homomorphic encryption schemes allow for operations on ciphertexts, enabling computations to be securely offloaded to third parties without leaking information about the encrypted data. Such schemes are *homomorphic* in that ciphertext operations correspond to plaintext operations: given encryption and decryption functions **Enc** and **Dec**, for a function f there exists a function f' such that $f(x) = \mathbf{Dec}(f'(\mathbf{Enc}(x)))$.

In a typical setting involving homomorphic encryption, a client encrypts their data with a private key and sends the ciphertext to a third-party server. The server performs operations over the ciphertext, and then sends the resulting ciphertext back to the client. The client can then decrypt the ciphertext to get the actual result of the computation.

We target modern lattice-based homomorphic encryption schemes such as BFV [40], BGV [15], and CKKS [26]. In these schemes, ciphertexts can encode many data elements at once. Thus we can treat ciphertexts as *vectors* of data elements. Homomorphic computations are expressed as addition and multiplication operations over ciphertexts. Addition and multiplication execute element-wise over encrypted data elements, allowing for SIMD processing: given ciphertexts $x = \mathbf{Enc}([x_1, x_2, \ldots, x_n])$ and $y = \mathbf{Enc}([y_1, y_2, \ldots, y_n])$, homomorphic addition $\oplus$ and multiplication $\otimes$ operate such that

$$\mathbf{Dec}(x \oplus y) = [x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n]$$
$$\mathbf{Dec}(x \otimes y) = [x_1 \times y_1, x_2 \times y_2, \ldots, x_n \times y_n].$$

There are analogous addition and multiplication operations between ciphertexts and plaintexts, which also have a vector structure. This allows computation over ciphertexts using data known to the server. For example, it is common to multiply a ciphertext with a plaintext *mask* consisting of 1s and 0s to zero out certain slots of the ciphertext.

Along with addition and multiplication, *rotation* facilitates data movement, cyclically shifting the slots of data elements by a specified amount. For example,

$$\mathbf{Dec}(\mathrm{rot}(-1, x)) = [x_2, x_3, \ldots, x_n, x_1] \qquad \mathbf{Dec}(\mathrm{rot}(2, x)) = [x_{n-1}, x_n, x_1, \ldots, x_{n-2}].$$

3.1.1 Programmability Challenges

While vectorized homomorphic encryption presents a viable approach to secure computation, there are many challenges to developing programs that use it. Such challenges include the lack of support for data-dependent control flow that forces programs to be written in "circuit" form; the selection of cryptographic parameters that

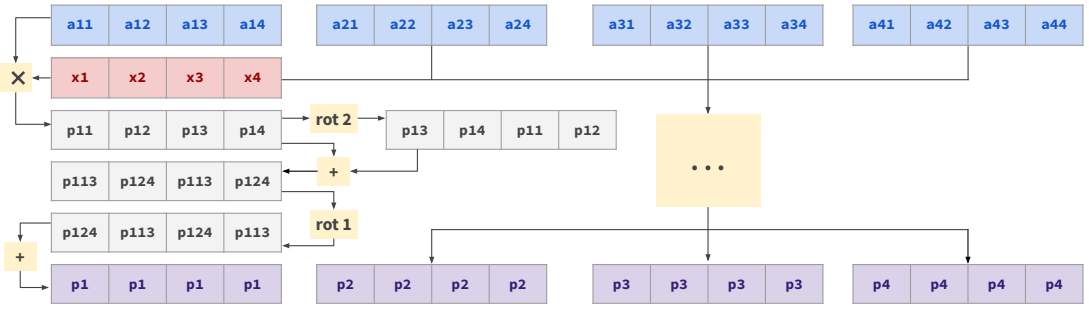

Figure 3.1: Row-wise layout for matrix multiplication.

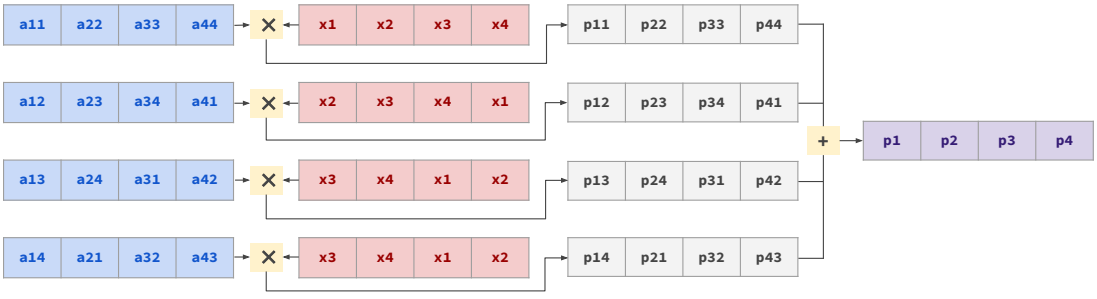

Figure 3.2: Diagonal layout for matrix multiplication.

are highly sensitive to the computations being executed; the management of ciphertext noise; and the interleaving of low-level "ciphertext maintenance" operations with computations [103].

Here we focus on the challenge of writing vectorized programs that use the SIMD capability of HE schemes. Efficiently vectorized HE programs are very different from programs in other regimes supporting SIMD. We now highlight some of the novelties of vectorizing in the HE regime.

Very long vector widths. Vector widths in HE ciphertexts are large powers of two—on the order of thousands when the scheme's parameters are set to appropriate security

levels [4]. To take advantage of such a large number of slots, often times HE programs are structured in counterintuitive ways. For example, the convolution kernel in Gazelle [58] applies a filter to all output pixels simultaneously. COPSE [73] evaluates decision forests by evaluating *all* branches at once and then applies masking to determine the right classification label for an input.

Limited data movement. Although ciphertexts can be treated as vectors, they have a very limited interface. In particular, one cannot index into a ciphertext to retrieve individual data elements. All operations are SIMD and compute on entire ciphertexts at once. Thus expressing computation that operates on individual data elements as ciphertext operations can be challenging. One might consider naive approaches to avoid such difficulties; for example, ciphertexts can be treated as single data elements by only using their first slot. Failure to restructure programs to take advantage of the SIMD capability of HE, however, exacts a steep performance hit: in many cases, orders of magnitude in slowdown [104].

So in practice, data elements must be packed in ciphertexts to write efficient HE programs. However, packing creates new problems: if an operation requires data on different slots, ciphertexts must be rotated to align the operands. One is thus forced to interleave data movement and computation, but determining how to schedule these together efficiently can be difficult.

Because rotation operations provide limited data movement, the initial data layout in ciphertexts has a great impact on the efficiency of HE programs. One layout might aggressively pack data to minimize the number of ciphertexts the client needs to send and also minimize the computations the server needs to perform, but might require too many rotations; another layout might not aggressively pack data into ciphertexts

to avoid the necessity of data movement operations, but might force greater client communication and the server to perform more computations.

Example: Matrix–vector multiplication. To illustrate the challenges of developing vectorized HE programs, consider the two implementations of matrix–vector multiplication. In both a matrix `a` is multiplied with a vector `x`; the vectors containing data elements from `a` are in blue, while the vectors containing data elements from `x` are in red; the output vectors of the multiplication are in purple.

Figure 3.1 shows a row-wise layout for the program, where the each of the blue vectors represents a row from `a`. A single red vector contains the vector `x`. The figure shows the computation of a dot product for one row of `a` and `x`; First, the vectors are multiplied, and then the product vector is rotated and added with itself multiple times to compute the sum. This pattern, which we call *rotate-and-reduce*, is common in the literature [31, 104, 62] and it exploits it allows for computing reductions in a logarithmic number of operations relative to the number of elements.[2] Here 4 elements can be summed with 2 rotations and 2 additions. The row-wise layout results in the dot product outputs to be spread out in 4 ciphertexts, which can preclude further computation (they cannot be used as a packed vector in another matrix–vector multiplication, say) and induce a lot of communication if the server sends these outputs to the client.

Figure 3.2 shows the "generalized diagonal" layout from Halevi and Shoup [53]. Here the vectors contain diagonals from array `a`: the first vector contains the main diagonal; the second vector contains a diagonal shifted to the right and wrapped around; and so on. These vectors are then multiplied with the vector containing `x`, but rotated

[2]The pattern is sometimes called "rotate-and-sum," but it clearly also applies to products as well. It requires the dimension size to be a power of two and the reduction operator to be associative and commutative, which is true for addition and multiplication.

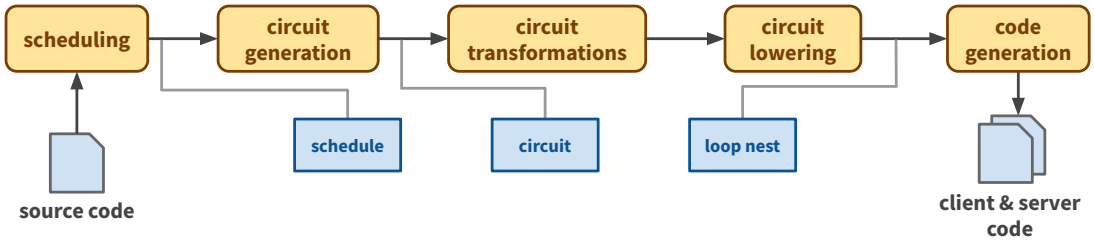

Figure 3.3: Viaduct-HE compiler architecture.

an appropriate number of slots. The layout of the product vectors allow the sum to be computed simply by adding the vectors together, as the product elements for different rows are packed in the same vector but elements for the same row are "exploded" along multiple vectors.

In total, the diagonal layout requires only 3 rotations and 3 additions, compared to the 8 additions and 8 rotations required by the row-wise layout. Additionally, the outputs are packed in a single vector, which can be convenient for further computations (it can be used as input to another matrix–vector multiplication) or for returning results to the client with minimal communication.

3.2 Compiler Overview

Figure 3.3 shows the architecture of the Viaduct-HE compiler. It takes as input an array program and generates both code run by the client, which sends inputs and receives program results, and by the server, which performs the computations that implement the source program. The compiler has well-defined extension points to control different aspects of the compilation process. To describe this process in detail, we consider the compilation of a program that computes the distance of a client-provided

```
input a: [4, 4] from server
input x: [4] from client
for j: 4 {
  sum(for i: 4 {
    (a[j][i] - x[i]) *
    (a[j][i] - x[i])
  })
}
```

Figure 3.4: Source code of distance program.

```
let out =
  sum_vec(i: 4,
    (at - rot(i, xt)) *
    (at - rot(i, xt)))
```

Figure 3.5: Circuit representation of distance program.

```
at    = PlaintextArray([4])
tmp0  = Vec(a, Roll(1,0), [0,0], [(2,0,0,{(0,1)}])
at[0] = encode(tmp0)
...
xt    = CiphertextArray([4])
xt[0] = Vec(x, Id, [0], [(4,0,0,{(0,1)})])
...
out   = CiphertextArray([])
for i in range(4) {
  inst1 = rot(C,i,xt[i])
  inst2 = sub(CP,inst1,at[i])
  inst3 = mul(C,inst2,inst2)
  out = add(C,out,inst3)
}
```

Figure 3.6: Loop-nest representation of distance program.

```
a = Array[4][4];
x = Array[4];
out = Array[4]
for (j = 0; j < 4; j++) {
  acc = 0;
  for (i = 0 ; i < 4; i++) {
    acc += (a[j][i] - x[i]) * (a[j][i] - x[i]);
  }
  out[j] = acc;
}
```

Figure 3.7: Distance program in a traditional imperative language.

point (x) against a list of test points known to the server (a).

3.2.1 Source Language

Figure 3.4 shows the source code for the distance program. It specifies that a is a 2D array provided as input by the server, with an extent of 4 on both dimensions; similarly, x is a 1D array provided as input by the client, with an extent of 4. Thus a is assumed to be known to the server and thus is in plaintext, while x is in ciphertext since it comes from the client. The two **for** nodes each introduce a new dimension to the output array; they also introduce the index variables i and j, which are used to index into the input arrays a and x. The dimension introduced by the inner **for** node is reduced with the sum operator, so the output array has one dimension. Conceptually, the program computes the distance of x from the rows of a, each of which represents a point. An equivalent implementation in a traditional imperative language would look like the following program on Figure 3.7.

Figure 3.8 defines the abstract syntax for the source language. Programs consist of a sequence of inputs and let-bound expressions followed by an output expression whose

Integer $z \in \mathbb{Z}$ Natural $n \in \mathbb{N}$ Index variable i, j, k Array variable a, b, c

Shape	sh	::=	$[n_1, \ldots, n_d]$
Party	pt	::=	**client** | **server**
Operator	$\odot$	::=	$+ \mid - \mid \times$
Index	in	::=	$i \mid z \mid in \odot in$
Expression	e	::=	$z \mid ie \mid e \odot e \mid \mathbf{reduce}_{\odot,n}(e) \mid \mathbf{for}\ i : n\ \{e\}$
Index Expression	ie	::=	$a \mid ie[in]$
Statement	s	::=	$\mathbf{let}\ a = e\ \mathbf{in}\ s \mid \mathbf{input}\ a : sh\ \mathbf{from}\ pt\ \mathbf{in}\ s \mid e$

Figure 3.8: Abstract syntax for the source language.

result the server sends to the client. Expressions uniformly denote arrays; scalars are considered zero-dimensional arrays. The expression **input** $a : sh$ **from** pt **in** e denotes an array with shape sh received as input from pt, which is either the client or the server. Input arrays from the client are treated as ciphertexts, while input arrays from the server are treated as plaintexts. Operation expression $e_1 \odot e_2$ denotes an element-wise operation over equal-dimension arrays denoted by e_1 and e_2, while reduction expression $\mathbf{reduce}_{\odot,n}(e)$ reduces the n-th dimension of the array denoted by e using $\odot$.

The expression **for** $i : n\ \{e\}$ adds a new outermost dimension with extent n to the array denoted by e, while $e[in]$—also referred throughout as an *indexing site*—indexes the outermost dimension of an array. Only array variables, introduced by inputs or let-bindings, can be indexed. The compiler also imposes some restrictions on indexing expressions. Particularly, index variables cannot be multiplied together (e.g. `a[i*j]`), as compiler analyses assume that the dimensions of indexed arrays are traversed with constant stride.

3.2.2 Scheduling

The source program is an abstract representation of computation over arrays; it represents the *algorithm*—the *what*—of an HE program. The vectorization *schedule*—*how* data will be represented by ciphertext and plaintext and how computations will be performed by HE operations—is left unspecified by the source program. Because its source language is array-oriented, the vectorization schedules for Viaduct-HE programs have a simple representation, allowing the compiler to manipulate such schedules and search for efficient ones during its scheduling stage. The compiler provides extension points to control both how the search space of schedules is explored and how the cost of schedules are assessed.

Like matrix multiplication, the distance program can be given a row-wise layout and a diagonal layout. The diagonal layout similarly requires less rotation and addition operations. The scheduling stage of the compiler can search for these schedules and assess their costs.

3.2.3 Circuit Representation

Once an efficient schedule has been found, the circuit generation stage of the compiler uses it to translate the source program into a circuit representation. The circuit representation represents information about the ciphertexts and plaintexts required in an HE program, as well as operations to be performed over these, at a very abstract level. The compiler has circuit transformation stages that leverage the algeraic properties of circuits to rewrite them into more efficient forms. Circuits are designed to facilitate optimization: a single circuit expression can represent many computations, so circuit

rewrites can optimize many computations simultaneously.

Figure 3.5 shows the circuit representation for the distance program with the diagonal layout. The `sum_vec` operation represents a summation of 4 different vectors together into one vector. The 4 vectors each represent a the result of a squared difference computation between vector containing a generalized diagonal of array `a` (represented by the variable **`at`**) and a rotated vector containing array `v` (represented by the variable `xt`).

3.2.4 Loop-nest Representation

Circuits represent HE computations at a very high level. This makes the circuit representation amenable to optimization, but makes generation of target code difficult. After circuit programs have been optimized, the circuit lowering stage of the compiler translates circuit programs into a "loop-nest" representation. Loop-nest programs are imperative programs that are much closer in structure to target code. Once in the loop-nest representation, the code generation stage of the compiler generates target code using a *back end* for a specific HE library. The compiler can generate code for a different HE library just by swapping out the back end it uses. Back ends only need to translate loop-nest programs to target code, so adding support for new back ends is straightforward.

Figure 3.6 shows the loop-nest representation for the distance program. It contains code to explicitly fill in the variables **`at`** and `xt` with the vectors that will be used in computations. The summation is now represented as an explicit **`for`** loop that accumulates squared distance computations in an **`out`** variable.

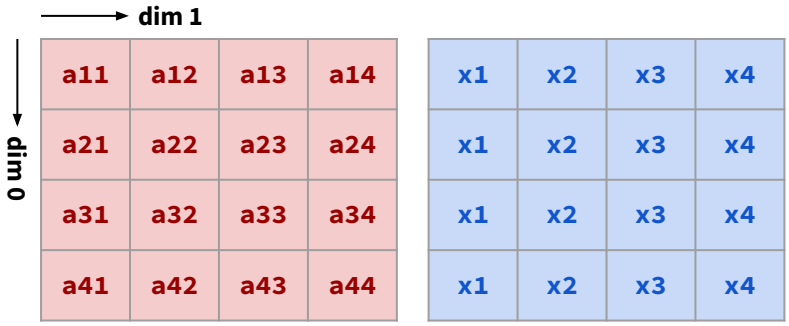

Figure 3.9: Array traversals in the distance program.

3.3 Scheduling

The scheduling stage begins by first translating source programs into an intermediate representation that eliminates explicit indexing constructs. From there, the compiler generates an initial schedule and explores the search space of vectorization schedules.

3.3.1 Index-free Representation

The index-free representation is similar to the source language, except that **`for`** nodes are eliminated and indexing sites are replaced with pair (is, at) of a unique identifier (is) and an *array traversal* (at) that summarizes the contents of the array denoted by the indexing site. Array traversals are arrays generated from indexing another array. Figure 3.9 shows the array traversals in the distance program. The traversal in red is from indexing array `a`; the traversal in blue is from indexing array `x`. Note that the dimension 0 is introduced by the **`for`** `j` node in the source program, while dimension 1 is introduced by the **`for`** `i` node. The traversal of array `x` repeats along dimension 0 because it is not indexed by `j` and thus does not change along that dimension.

Formally, array traversals have three components: the name of the indexed array; the integer *offsets* at which the traversal begins, defined by a list of integers with a length equal to the number of dimensions of the indexed array; and a list of *traversal dimensions* (td). We write $\mathrm{a}(z_1, \ldots, z_m)[td_1, \ldots, td_n]$ to denote a n-dimensional traversal of an m-dimensional array. Array traversals can define positions that are out-of-bounds; for example, offsets can be negative even though all index positions in an array start at 0.

Each traversal dimension has an *extent* specifying its size and a set of *content dimensions* that specify how the dimensions traverses the indexed array. Content dimensions have a *dimension index* and a *stride*. For example, a traversal dimension $(4, \{0 :: 2\})$ defines a traversal of an array along its zeroth dimension that spans 4 elements, where only every other element is traversed (i.e. the stride is 2). Traversal dimensions can have empty content dimension sets, which means that the array traversal does not vary along the dimension. We call these traversal dimensions *empty*.

For example, the index-free representation for the distance program is

```
1 sum(1, ((at1,atr) - (xt1,xtr)) * (((at2,atr) - (xt2,xtr))).
```

Variables `at1` and `at2` represent indexing sites with traversal `atr` that indexes array `a`; variables `xt1` and `xt2` represent indexing sites with traversal `xtr` that indexes array `x`. The array traversals denoted by these indexing sites is as follows:

$$\mathrm{atr} = \mathrm{a}(0,0)[(4, \{0 :: 1\}), (4, \{1 :: 1\})] \qquad \mathrm{xtr} = \mathrm{x}(0)[(4, \{\}), (4, \{0 :: 1\})].$$

The traversal `atr` defines a 4x4 array where dimension 0 traverses dimension 0 of input array `a` with stride 1, and dimension 1 traverses dimension 1 of array `a` with stride 1. Meanwhile, the traversal `xtr` also defines a 4x4 array, but its dimension 0 is empty and its dimension 1 traverses the only dimension of input array `x` with stride 1.

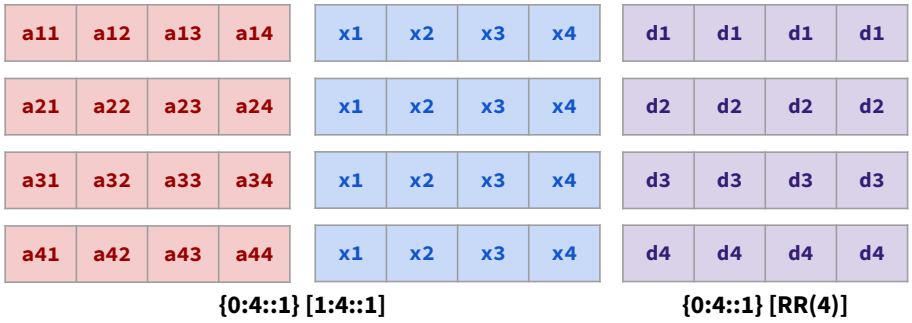

Figure 3.10: Row-wise layout. Induces 5 input vectors, 4 output vectors, 8 additions, 8 rotations (2 adds and rotates per vector with rotate-and-reduce).

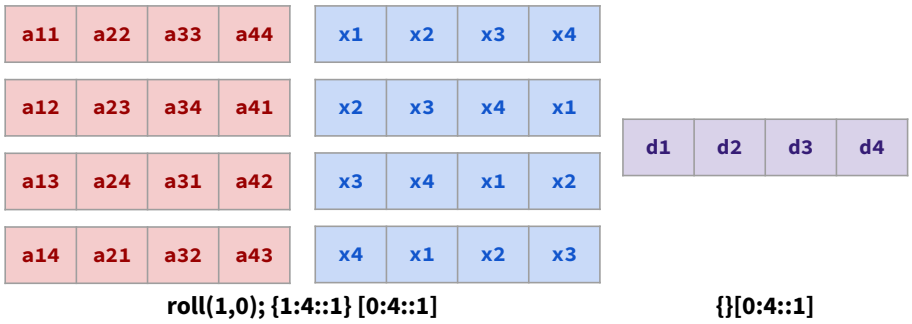

Figure 3.11: "Diagonal" layout. Induces 5 input vectors, 1 output vector, 3 additions, 3 rotations.

3.3.2 Representing Schedules

Schedules define a *layout* for the array traversals denoted by each indexing site in the index-free representation. The layout determines how an array traversal is represented as a set of vectors. One can think of layouts as a kind of traversal of array traversals themselves. Because of this, layouts are defined similarly to array traversals, except they do not specify offsets, as layouts always have offset 0 along every traversal dimension.

Layouts are built from *schedule dimensions* which denote some part of an array traversal. We write the syntax $i : n :: s$ for a schedule dimension with dimension index i, extent n, and stride s. For example, a schedule dimension $0 : 4 :: 2$ defines a 4-element section of an array traversal along its zeroth dimension that contains only every other element (i.e. the stride is 2).

Concretely, layouts consist of the following: (1) a set of *exploded* dimensions; (2) a list of *vectorized* dimensions; and (3) a preprocessing operation. Exploded dimensions define parts of the array traversal that will be laid out in *different* vectors, while vectorized dimensions define parts that will be laid out in *every* vector. The ordering of vectorized dimensions defines their ordering on a vector: the beginning of the list defines the outermost vectorized dimensions, while the end defines the innermost vectorized dimensions. Preprocessing operations change the contents of the array traversal before being laid out into vectors, which allow for the representation of complex layouts. We write $p\{ed_1, \ldots, ed_m\}[vd_1, \ldots, vd_n]$ to denote a schedule with preprocessing p, m exploded dimensions (ed) and n vectorized dimensions (vd). When p is the identity preprocessing operation, it is often omitted from the schedule.

Figure 3.10 and Figure 3.11 show two different schedules for the distance program. The vectors of `at1` and `at2` are in red, while the vectors of `xt1` and `xt2` are in blue. Their respective layouts are given below the vectors. Finally, the vectors of the distance program's output is in purple, and the output layout is given below. Note that array traversals for **`at`** and `xt` must have the same layout since their arrays are multiplied together, and operands of element-wise operations must have the same layout.

Figure 3.10 represents a row-wise layout, where the entirety of dimension 1 of **`at`** and `xt` are vectorized while the entirety dimension 0 is exploded into multiple vectors. Thus traversal **`at`** is represented by 4 vectors, one for each of its rows; since traver-

sal `xt` has 4 equal rows, it is represented by a single vector. Meanwhile, Figure 3.11 represents a "diagonal" layout; it is similar to a column-wise layout where each vector contains a column, but the `roll` preprocessing operation rotates the rows along the columns, where the rotation amount progressively increases. As discussed in §3.1, a similar diagonal layout was originally specified in Halevi and Shoup [53] as an efficient implementation of matrix–vector multiplication, but here we see that the schedule abstraction can capture its essence, allowing the compiler to generalize and use it for other programs.

Note that exploded dimensions have a name associated with them; in the above syntax, the name for exploded dim i is d_i. These names are used to uniquely identify vectors induced by the layout. During circuit generation, the names of exploded schedule dimensions will be used as variables that parameterize circuit expressions.

Preprocessing. A preprocessing operation in a layout transforms an array traversal before laying it out into vectors. Formally, a preprocessing operation is a permutation over elements of the array traversal. Thus we can think of preprocessing operations as functions from element positions to element positions. Given an n-dimensional array traversal at, applying preprocessing operation p over at defines a new traversal such that the element at position $x_1, \ldots, x_n$ is the element at position $p(x_1, \ldots, x_n)$ of at. For example, the identity preprocessing operation is the trivial permutation that maps element positions to themselves: $\text{id} = \lambda(x_1, \ldots, x_n).(x_1, \ldots, x_n)$. Given that both dimensions i and j of an array traversal both have extent n, we can define the `roll` preprocessing operation as follows:

$$\text{roll}(a, b) = \lambda(x_1, \ldots, x_a, \ldots, x_b, \ldots, x_n).(x_1, \ldots, x_a + x_b \,\%\, n, \ldots, x_b, \ldots x_n).$$

Applying layouts. When applied to an array traversal, a layout generates a set of vectors that contain parts of the array indexed by the traversal. Formally, a vector contains four components: the name of the indexed array; a preprocessing operation; a list of integer offsets; and a list of traversal dimensions. As with preprocessing in a layout, a preprocessing operation in a vector transforms an array before its contents are laid out in the vector. We write $a.p(z_1, \ldots, z_m)[vtd_1, \ldots, vtd_n]$ to denote a vector indexing an m-dimensional array a with preprocessing p and n traversal dimensions (vtd_i). Again p is usually elided when it is the identity preprocessing operation. Vector traversal dimensions are similar to array traversal dimensions, except they also track elements that are out-of-bounds. A vector can have out-of-bounds values either because its dimensions extend beyond the extents of the indexed array or the extents of the array traversal from which it is generated. We write $(n, obl, obr, \{cd_1, \ldots, cd_m\})$ to denote a vector traversal dimension with extent n, m content dimensions, a left out-of-bounds extent obl, and a right out-of-bounds extent obr. The left out-of-bounds and right out-of-bounds extents count the number of positions in a dimension that are out-of-bounds to the left and right of the in-bounds positions respectively. The compiler enforces the semantics that out-of-bounds values have value 0.

Given a layout with n exploded dimensions each with extent n_i, applying the layout to an array traversal generates $\prod n_i$ vectors, one for each distinct combination of positions that can be defined along exploded dimensions. We call each such combination a *coordinate*. For example, when applied to array traversal **`at`**, the diagonal layout for the distance program generates 4 vectors, one for each distinct positions that the

exploded dimension named i can take:

$$\{i \mapsto 0\} \mapsto \text{a.roll}(1,0)(0,0)[(4,0,0,\{0 :: 1\})]$$

$$\{i \mapsto 1\} \mapsto \text{a.roll}(1,0)(0,1)[(4,0,0,\{0 :: 1\})]$$

$$\{i \mapsto 2\} \mapsto \text{a.roll}(1,0)(0,2)[(4,0,0,\{0 :: 1\})]$$

$$\{i \mapsto 3\} \mapsto \text{a.roll}(1,0)(0,3)[(4,0,0,\{0 :: 1\})].$$

This represents the same vectors for traversal **`at`** visually represented in Figure 3.11.

3.3.3 Searching for Schedules

To search for an efficient schedule for the program, the scheduling stage begins with an initial schedule where the layouts for all indexing sites contain only exploded dimensions. Thus in this schedule elements of arrays are placed in individual vectors. While very inefficient, the initial schedule can be defined for any program. To explore the search space of schedules, the scheduling stage uses a set of *schedule transformers* that take a schedule as input and returns a set of "nearby" schedules. To assess both the validity of a schedule visited during search, the compiler attempts to generate a circuit from the schedule. If a circuit is successfully generated, it is applied to a *cost estimator* function to determine the cost of the schedule.

The schedule transformers in the prototype implementation of the compiler include the following.

Vectorize dimension transformer. This transformer takes an exploded dimension from a layout and vectorizes it:

$$p\{\ldots,(d_a)\ i_a : n_a :: s_a,\ldots\}[\ldots] \rightsquigarrow p\{\ldots\}[\ldots,i_a : n_a :: s_a].$$

Importantly, this transformer only generates vectorized dimensions with extents that are powers of two; if the exploded dimension is not a power of two, the transformer will round up the vectorized dimension's extent to the nearest one. The transformer imposes this limit on vectorized dimensions to simplify reasoning about correctness: vectors only wrap around correctly when their size divides the slot counts of ciphertexts and plaintexts without remainder, and these slot counts are always powers of two. This limitation also allows the circuit generation stage to uniformly use the rotate-and-reduce pattern.

Tiling transformer. This transformer takes an exploded dimension and *tiles* it into an outer dimension and an inner dimension. That is, given that extent n_a can be split into n tiles each of size t (i.e., $e_a = tn$), it performs the following transformation:

$$p\{\ldots, (d_a)\ i_a : n_a :: s_a, \ldots\}[\ldots] \rightsquigarrow p\{\ldots, (d'_a)\ i_a : t :: s_a, (d''_a)\ i_a : n :: s_a t, \ldots\}[\ldots]$$

where d'_i and d''_i are fresh exploded dimension names.

Roll transformer. This transformer applies a roll preprocessing operation to a layout:

$$\text{id}\{\ldots, (d_a)\ i_a : n :: s_a, \ldots\}[i_b : n :: s_b, \ldots] \rightsquigarrow$$
$$\text{roll}(a, b)\{\ldots, (d_a)\ i_a : n :: s_a, \ldots\}[i_b : n :: s_b, \ldots].$$

The transformer only applies when dimension a is exploded, dimension b is the outermost vectorized dimension and their extents match. The following conditions must also hold:

- the traversal dimensions a and b are not tiled;
- a and b have content dimension sets with size at most 1;

- if a or b has content dimension set $\{d :: s\}$, the extent of d must be the same as the extents of a and b and stride s must equal 1;
- the dimension in the indexed array traversed by a and b, if any, must not be traversed in other dimensions.

These conditions ensure that layouts with roll preprocessing can be materialized.

Epochs. The search space for vectorization schedules is large. To control the amount of time that scheduling takes, the search is staggered into *epochs*. During an epoch, the configuration of schedule transformers is fixed such that only a subset of the search space is explored. When no more schedules can be visited, the epoch ends; a new epoch then begins with the schedule transformers updated to allow exploration of a bigger subset of the search space. The compiler runs a set number of epochs, after which it uses the most efficient schedule found to proceed to later stages of compilation.

The prototype implementation of the Viaduct-HE compiler uses epochs to control how schedule dimensions are split by the tiling transformer, which is the main cause of search space explosion. The tiling transformer gradually increases the number of schedule dimensions it splits as the number of scheduling epochs increase.

3.4 Circuit Generation

The circuit generation stage takes a schedule and index-free program as input and attempts to generate a circuit program. The design of the circuit representation reflects the fact that many computations in HE programs are structurally similar. Thus a circuit expression denotes not just a single HE computations, but rather a family of HE

Dimension variable $d \in \mathcal{D}$ Array name a Vector v
Plaintext var φ_p Ciphertext var φ_c Offset var φ_o
Circuit value map $cm_\tau \in (\mathcal{D} \times \cdots \times \mathcal{D} \rightharpoonup \mathbb{N} \times \cdots \times \mathbb{N}) \rightharpoonup \tau$

Offset	oe	$::=$	$d \mid z \mid oe \odot oe \mid \varphi_o$
Expression	ce	$::=$	$\varphi_p \mid \varphi_c \mid z \mid ce \odot ce$
		$\mid$	$\textbf{rot}(oe, ce) \mid \textbf{reduce-vec}_\odot(d : n, ce)$
Statement	cs	$::=$	$\textbf{let}\ a : [d_1 : n_1, \ldots, d_n : n_n] = ce \mid cs; cs \mid \textbf{skip}$
Object	$co \in \mathcal{O}$	$::=$	$\text{Const}(z) \mid \text{Mask}(\overline{(n, n, n)}) \mid \text{Vec}(v)$
Registry	cr	$::=$	$\varphi_o \mapsto cm_\mathbb{Z}, cr \mid \varphi_p \mapsto cm_\mathcal{O}, cr \mid \varphi_c \mapsto cm_\mathcal{O}, cr \mid \cdot$

Figure 3.12: Abstract syntax for circuit programs.

Array Materializer A Layout ℓ
Index-free expression fe Index-free statement fs
Schedule dimension sd Exploded dimension ed

Output Vectorized Dimension	ovd	$::=$	$i : n :: s \mid \text{R}(n) \mid \text{RR}(n)$
Output Layouts	$o\ell$	$::=$	$* \mid p\{ed_1, \ldots, ed_m\}[ovd_1, \ldots, ovd_n]$
Schedule	Σ	$::=$	$\Sigma, is : \ell \mid \cdot$
Input Context	Γ	$::=$	$\Gamma, a : sh \mid \cdot$
Expression Context	Δ	$::=$	$\Delta, a : (sh, o\ell) \mid \cdot$

Figure 3.13: Syntax for circuit generation.

computations. Expressions are parameterized by *dimension variables*, and an expression represents a different computation for each combination of values (coordinates) these variables take.

Figure 3.12 shows the abstract syntax for circuit programs. A circuit program consists of a sequence of let statements that bind the results of expressions to array names; the last of these statements defines a distingished output array (out) whose results will be sent to the client. A statement $\textbf{let}\ a : [d_1 : n_1, \ldots, d_n : n_n] = ce$ declares an array a whose contents is computed by expression ce. Note that ce is parameterized by dimension variables d_i each with extent n_i, which means that ce represents $\prod n_i$ different

$\boxed{A; \Sigma; \Gamma; \Delta \vdash fe \rightsquigarrow ce : (sh, o\ell)}$ $\boxed{A; \Sigma; \Gamma; \Delta \vdash fs \rightsquigarrow cs}$

$* <: sh$ $* <: o\ell$ $p\{\ldots\}[\text{RR}(n), ovd_2, \ldots, ovd_n] <: p\{\ldots\}[ovd_2, \ldots, ovd_n]$

CGen-Literal

$$\frac{}{A; \Sigma; \Gamma; \Delta \vdash z \rightsquigarrow z : (*, *)}$$

CGen-Op

$$\frac{A; \Sigma; \Gamma; \Delta \vdash fe_1 \rightsquigarrow ce_1 : (sh, o\ell) \quad A; \Sigma; \Gamma; \Delta \vdash fe_2 \rightsquigarrow ce_2 : (sh, o\ell)}{A; \Sigma; \Gamma; \Delta \vdash fe_1 \odot fe_2 \rightsquigarrow ce_1 \odot ce_2 : (sh, o\ell)}$$

CGen-Input-Index

$$\frac{\Sigma(is) = \ell \quad \Gamma(a) = sh_a \quad a = \textsc{tr-array}(at) \quad sh = \textsc{tr-shape}(at) \quad \textsc{materialize-input}(A, sh_a, at, \ell) \rightsquigarrow ce}{A; \Sigma; \Gamma; \Delta \vdash (is, at) \rightsquigarrow ce : (sh, \ell)}$$

CGen-Expr-Index

$$\frac{\Sigma(is) = \ell \quad \Delta(a) = (sh_a, o\ell_a) \quad a = \textsc{tr-array}(at) \quad sh = \textsc{tr-shape}(at) \quad \textsc{materialize-expr}(A, sh_a, o\ell_a, at, \ell) \rightsquigarrow ce}{A; \Sigma; \Gamma; \Delta \vdash (is, at) \rightsquigarrow ce : (sh, \ell)}$$

CGen-Reduce

$$\frac{sh_1 = [n_0, \ldots, n_{n-1}, n_n, n_{n+1}, \ldots, n_{d-1}] \quad sh_2 = [n_0, \ldots, n_{n-1}, n_{n+1}, \ldots, n_{d-1}] \quad A; \Sigma; \Gamma; \Delta \vdash fe \rightsquigarrow ce : (sh_1, o\ell_1) \quad \textsc{reduce-layout}(n, o\ell_1) \rightsquigarrow (o\ell_2, dl)}{A; \Sigma; \Gamma; \Delta \vdash \textbf{reduce}_{\odot,n}(fe) \rightsquigarrow \textsc{gen-reduce}_{\odot}(ce, dl) : (sh_2, o\ell_2)}$$

CGen-Input

$$\frac{A; \Sigma; \Gamma, a : sh; \Delta \vdash fs \rightsquigarrow cs}{A; \Sigma; \Gamma; \Delta \vdash \textbf{input}\ a : sh\ \textbf{from}\ pt\ \textbf{in}\ fe \rightsquigarrow cs}$$

CGen-Let

$$\frac{o\ell = p\{(d_1)\ i_1 : n_1 :: s_1, \ldots, (d_n)\ i_n : n_n :: s_n\}[\ldots] \quad A; \Sigma; \Gamma; \Delta \vdash fe \rightsquigarrow ce : (sh, o\ell) \quad A; \Sigma; \Gamma; \Delta, a : (sh, o\ell) \vdash fs \rightsquigarrow cs}{A; \Sigma; \Gamma; \Delta \vdash \textbf{let}\ a = fe\ \textbf{in}\ fs \rightsquigarrow \textbf{let}\ a : [d_1 : n_1, \ldots, d_n : n_n] = ce; cs}$$

Figure 3.14: Rules for circuit generation.

computations, one for each distinct combination of values that the variables can take. Because expressions can vary depending on the coordinates their dimension variables take, circuit programs are accompanied by a *circuit registry* data structure that records information about the exact values expressions take at a particular coordinate.

Circuit expressions include literals (z) and operations ($ce_1 \odot ce_2$) as in source programs. Expression **rot**(oe, ce) rotates the vector denoted by ce by an offset oe. Offsets can include literals, operations, index variables, and offset variables (φ_o); the latter two allows rotation amounts to vary depending on the values of in-scope dimension variables. The value of an offset variable at a particular coordinate is defined by a map that is stored in the registry that comes with the circuit program. Ciphertext (φ_c) and plaintext (φ_p) variables define a family of ciphertext and plaintext vectors respectively. Like offset variables, the exact vector these variables represent at a particular coordinates is defined by a map in the registry. These vectors can contain parts of input arrays and result arrays of prior expressions; additionally, plaintext variables can also represent constant vectors, which contain the same value in all of its slots, and mask vectors, which can be multiplied to another vector to zero out some of its slots. Masks are defined by a list of dimensions $[(n_1, lo_1, hi_1), \ldots, (n_n, lo_n, hi_n)]$, where n_i is the extent of the dimension i and $[lo_i, hi_i]$ is the *defined interval* for dimension i. The mask has value 1 in slots within defined intervals and 0 in slots outside of defined intervals.

Finally, the expression **reduce-vec**$_\odot(d : n, ce)$ defines a computation where multiple vectors are reduced to a single vector with operation $\odot$. If the reduction expression is parameterized by dimension variables $d_1, \ldots, d_n$ with extents $n_1, \ldots, n_n$, then the expression represents $\prod n_i$ different vectors, each of which were computed by reducing d vectors together. Thus the expression ce is parameterized by variables $d_1, \ldots, d_n, d$.

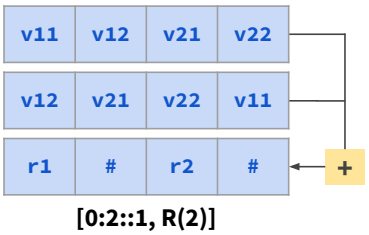

Figure 3.15: A reduced vectorized dimension.

3.4.1 Translation Rules for Circuit Generation

The translation into the circuit representation is mostly standard across programs, with the exception of the translation of indexing sites. The compiler uses a set of *array materializers* that lower the array traversals denoted by indexing sites into vectors and circuit operations according to a specific layout. We discuss them in detail in §3.4.2.

Figure 3.13 and Figure 3.14 shows the rules for generating a circuit program from the index-free representation. The judgment defines both the translation to a circuit as well as the conditions that must hold for the translation to be successful. The expression translation judgment has form $A; \Sigma; \Gamma; \Delta \vdash fe \rightsquigarrow ce : (sh, \ell)$, which means that given an array materializer configuration A, schedule Σ, input context Γ, and expression context Δ, the index-free expression fe can be translated to circuit expression ce, where the computation defined by ce has shape sh and output layout ℓ. The output layout defines how the results are laid out in vectors. The input context defines the shapes of input arrays in scope, while the expression context defines the shape and output layout of let-bound arrays in scope. The statement translation judgment $A; \Sigma; \Gamma; \Delta \vdash fs \rightsquigarrow cs$ has a similar form to expression translations. Additionally, judgment $sh_1 <: sh_2$ means that shape sh_1 can be coerced to shape sh_2, and similarly $o\ell_1 <: o\ell_2$ means that output layout $o\ell_1$ can be coerced to $o\ell_2$.

Output layouts. Note that output layouts are more general than the layouts defined by schedules. First, they can be the "wildcard" layout ($*$), which can be coerced into any layout. Second, vectorized dimensions can take other forms. A *reduced dimension* (R(n)) represents a dimension in a vector with extent n, but since it is reduced only the first position of the dimension has an array element; the rest of the dimension contain invalid values. A vectorized dimension becomes a reduced dimension when its contents are rotated-and-reduced. Figure 3.15 shows the output layout for vector with a starting layout of $[0 : 2 :: 1, 1 : 2 :: 1]$ after its inner dimension has been reduced.

When the outermost vectorized dimension is rotated-and-reduced, however, the elements of the dimension wrap around such that the result of the reduction repeats along the extent of the dimension. This can be seen in the row-wise layouts for matrix–vector multiplication (Figure 3.1) and the distance program (Figure 3.10). In that case, a vectorized dimension with extent n becomes a *reduced repeated* dimension (RR(n)). Output layouts with reduced repeated dimensions can be coerced into layouts that drop such dimensions.

Translations. The translations of literals (CGen-Literal) and operations (CGen-Op) are straightforward; CGen-Op additionally ensures that the operands have the same shape and output layout. The translations for indexing sites (CGen-Input-Index and CGen-Expr-Index) use the compiler's array materializer configuration A to lower an array traversal into a layout specified by the schedule. The functions tr-array and tr-shape return the indexed array and shape of an array traversal respectively; the functions materialize-input and materialize-expr are part of the interface of array materializers and, if successful, return a circuit expression representing the vectors of the array traversal in the required layout. The translations for statements (CGen-

INPUT, CGEN-LET, CGEN-OUTPUT) add array information to the context. Note that the translation for let statements additionally uses the exploded dimensions of the output layout of its body expression circuit as dimension variables to parameterize the circuit.

The translation of reduction expressions (CGEN-REDUCE) are more involved. Given $\textbf{reduce}_{\odot,n}(fe)$ and that fe is translated to ce, the output layout $o\ell_1$ of ce is transformed to an output layout $o\ell_2$ that reflects the reduction by the REDUCE-LAYOUT function, which returns $o\ell_2$ layout as well as the list of schedule dimensions (dl) in ℓ_1 that were reduced. Let i be the traversal dimension index referenced by a schedule dimension sd in ℓ_1. Then there are three possible cases:

- When $i < n$, then sd remains in ℓ_2 unchanged.
- When $i > n$, then sd remains in ℓ_2 but now references traversal dimension index $i - 1$.
- When $i = n$ and sd is exploded, it is removed from ℓ_2 entirely and added to the list of reduced schedule dimnesions dl. When $i = n$ and sd is vectorized with extent n, it is either replaced with a reduced dimension $\mathrm{R}(n)$ or a reduced repeated dimension $\mathrm{R}(n)$ depending on its position. sd is added to the list of reduced schedule dimensions dl along with its *block size* (b), which is intuitively the number of vector slots between array elements whose positions have different values of i. It is defined the product of the extents of the vectorized dimensions that come after sd in the layout. For example, if sd is the innermost vectorized dimension (i.e., it is the last element in a layout's list of vectorized dimensions), then $b = 1$. The block size is a parameter used by the rotate-and-reduce pattern that will compute the reduction for the vectorized dimension.

Note that REDUCE-LAYOUT fails when the preprocessing operation of the layout can-

$$
\begin{aligned}
\textsc{gen-reduce}_{\odot}(ce, []) &= ce \\
\textsc{gen-reduce}_{\odot}(ce, [(d)\ i : n :: s] + tl) &= \textsc{gen-reduce}_{\odot}(\textbf{reduce-vec}_{\odot}(d : n, ce), tl) \\
\textsc{gen-reduce}_{\odot}(ce, [(i : n :: s, b)] + tl) &= \textsc{gen-reduce}_{\odot}(\textsc{rot-reduce}_{\odot}(b, n, ce), tl) \\
\textsc{rot-reduce}_{\odot}(b, 1, ce) &= ce \\
\textsc{rot-reduce}_{\odot}(b, n, ce) &= \textsc{rot-reduce}_{\odot}(b, n/2, ce \odot \textbf{rot}(b(n/2), ce))
\end{aligned}
$$

Figure 3.16: Definition of GEN-REDUCE$_{\odot}$.

not be successfully transformed by the REDUCE-PREPROCESS function, which is specific to each preprocessing operation. Given identity preprocessing operation (id), REDUCE-PREPROCESS always succeeds and returns id unchanged. Meanwhile, given preprocessing roll(a, b) and reduced dimension index n REDUCE-PREPROCESS returns either id when $n = a$ or roll(a, b) when $b \neq n \neq a$. When $n = b$, REDUCE-PREPROCESS is not defined and fails. Intuitively, reducing dimension a transforms roll into id since it only changes the positions of elements along a. Meanwhile, reducing dimension b would reduce array elements together that originally had positions with different values for a before roll was applied, which is invalid.

Finally, the GEN-REDUCE$_{\odot}$ function, defined in Figure 3.16, generates the circuit expressions necessary to translate the reduction. It takes the list of reduced schedule dimensions generated by REDUCE-LAYOUT and for each schedule dimension either adds a reduce-vec expression to the circuit, if the dimension is exploded, or generates a rotate-and-reduce pattern, if the dimension is vectorized.

3.4.2 Array Materialization

Array materializers allow the compiler to customize how a layout is applied to an array traversal. They can be triggered to run only for certain array traversals and layouts, and thus can use specialized information about these to enable complex translations.

Array materializers implement two main functions. The MATERIALIZE-INPUT function materializes an array traversal indexing an input array. It takes as input the shape of the indexed array (sh_a), the array traversal itself (at), and the layout for the traversal specified by the schedule (ℓ). The MATERIALIZE-EXPR function materializes an array traversal indexing an array that is the output of a let-bound statement. It takes similar input to MATERIALIZE-INPUT with the addition of the output layout of the indexed array ($o\ell_a$).

Vector Derivation. The prototype implementation of the Viaduct-HE compiler has two array materializers. The first is the default materializer that is triggered on layouts with no preprocessing. When materializing traversals of input arrays, it attempts to minimize the number of input vectors required by *deriving* vectors from one another. When materializing traversals of let-bound arrays, it attempts to derive vectors of the traversal from the vectors defined by the output layout of the indexed array; materialization fails if some vector for the traversal cannot be derived.

Intuitively, a vector v_1 can be derived from another vector v_2 if all the array elements traversed by v_1 are contained in v_2 in the same relative positions, although rotation and masking might be required for the derivation. For example, consider the layout ℓ for traversal `kt` of 4x4 client input array `k`:

$$\mathrm{kt} = \mathrm{k}(0,0)[(2,\{0 :: 1\}),(4,\{0 :: 1\})] \qquad \ell = \{(i)\ 0 : 2 :: 1\}[1 : 4 :: 1].$$

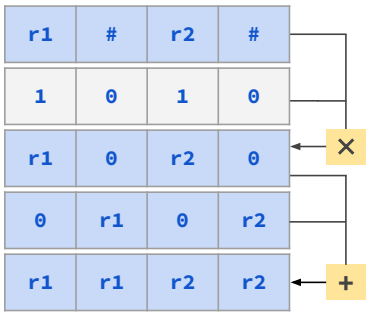

Figure 3.17: Clean-and-fill pattern.

Applying ℓ to `kt` yields two vectors:

$$\{i \mapsto 0\} \mapsto \mathrm{k}(1,0)[(3,0,1,\{0 :: 1\})] \quad \{i \mapsto 1\} \mapsto \mathrm{k}(0,0)[(4,0,0,\{0 :: 1\})].$$

Then the vector at $\{i \mapsto 0\}$ can be derived from the vector at $\{i \mapsto 1\}$ by rotating the latter by -1 and masking its 4th slot. The materializer then generates the circuit expression $\mathbf{rot}(\varphi_o, \varphi_c) \times \varphi_p$ for the `kt` and adds the following mappings to the registry:

$$\begin{aligned}
\varphi_c \mapsto &\{\{i \mapsto 0\} \mapsto \mathrm{Vec}(\mathrm{k}(0,0)[(4,0,0,\{0 :: 1\})]),\\
&\quad \{i \mapsto 1\} \mapsto \mathrm{Vec}(\mathrm{k}(0,0)[(4,0,0,\{0 :: 1\})])\}\\
\varphi_o \mapsto &\{\{i \mapsto 0\} \mapsto -1, \{i \mapsto 1\} \mapsto 0\}\\
\varphi_p \mapsto &\{\{i \mapsto 0\} \mapsto \mathrm{Mask}((4,0,2)), \{i \mapsto 1\} \mapsto \mathrm{Const}(1)\}.
\end{aligned}$$

Besides rotation and masking, if a vector has an empty dimension it can be derived from a vector that contains a reduced dimension in the same position using a "clean-and-fill" routine, seen in Figure 3.17 [3][33, Figure 1]. This is useful for deriving vectors of traversals that index let-bound arrays.

Roll Materializer. The other array materializer used by the Viaduct-HE compiler is specifically for layouts with a roll preprocessing operation. Given traversal dimensions a and b, let $a = (n, cd_a)$ and $b = (n, cd_b)$. There are three possible cases:

- **Case 1**: $cd_a = \{\}$. Then the contents of the array traversal do not change along a, so the roll preprocessing does not do anything. The traversal will be materialized as if the layout has no preprocessing.
- **Case 2**: $cd_a = \{i_a :: 1\}$ and $cd_b = \{i_b :: 1\}$. Then the traversal will be materialized as if the layout has no preprocessing, but the vectors generated will have preprocessing roll(i_a, i_b).
- **Case 3**: $cd_a = \{i_a :: 1\}$ and $cd_b = \{\}$. Then the traversal will be materialized as if the layout has no preprocessing, but only for vectors where $\{b \mapsto 0\}$. To materialize a vector at coordinate $c = \{\ldots, b \mapsto v, \ldots\}$ where $v \neq 0$, the vector at coordinate $c[b \mapsto 0]$ (i.e. c but with b set to 0) is rotated amount v.

Example. Consider the distance example from Figure 3.4 compiled with the diagonal layout. Traversals at and xt are defined as follows:

$$\text{at} = \text{a}(0,0)[(4,\{0 :: 1\}),(4,\{1 :: 1\})] \qquad \text{xt} = \text{x}(0)[(4,\{\}),(4,\{0 :: 1\})].$$

The diagonal layout is defined as roll$(1,0)\{(i)\ 1 : 4 :: 1\}[0 : 4 :: 1]$. Thus when we apply the layout to at, case 2 above holds. So the materializer returns ciphertext variable φ_c as the circuit expression representing at and φ_c is mapped to the following in the circuit registry:

$$\{i \mapsto 0\} \mapsto \text{a.roll}(1,0)(0,0)[(4,0,0,\{0 :: 1\})]$$
$$\{i \mapsto 1\} \mapsto \text{a.roll}(1,0)(0,1)[(4,0,0,\{0 :: 1\})]$$
$$\{i \mapsto 2\} \mapsto \text{a.roll}(1,0)(0,2)[(4,0,0,\{0 :: 1\})]$$
$$\{i \mapsto 3\} \mapsto \text{a.roll}(1,0)(0,3)[(4,0,0,\{0 :: 1\})].$$

When we apply the layout to xt, case 3 holds. Thus the materializer returns

$$ce + 0 = ce \qquad ce \times 1 = ce \qquad ce \times 0 = 0 \qquad ce_1 - ce_2 = ce_1 + (-1 \times ce_2)$$

$$ce_1 \times (ce_2 + ce_3) = (ce_1 \times ce_2) + (ce_1 \times ce_3)$$

$$\textbf{rot}(oe_1 + oe_2, ce) = \textbf{rot}(oe_1, \textbf{rot}(oe_2, ce))$$

$$\textbf{rot}(oe, ce_1) + \textbf{rot}(oe, ce_2) = \textbf{rot}(oe, ce_1 + ce_2)$$

$$\textbf{reduce-vec}_+(d : n, z \times ce) = z \times \textbf{reduce-vec}_+(d : n, ce)$$

$$\textbf{reduce-vec}_+(d_1 : n_1, \textbf{reduce-vec}_+(d_2 : n_2, ce)) = \textbf{reduce-vec}_+(d_2 : n_2, \textbf{reduce-vec}_+(d_1 : n_1, ce))$$

$$\frac{d \notin \text{dim-vars}(oe)}{\textbf{reduce-vec}_\odot(d : n, \textbf{rot}(oe, ce)) = \textbf{rot}(oe, \textbf{reduce-vec}_\odot(d : n, ce))}$$

Figure 3.18: Select identities for circuit optimization.

$\textbf{rot}(i, \varphi_c)$ as the circuit expression representing xt and φ_c is mapped to the following in the circuit registry:

$$\{i \mapsto 0\} \mapsto \text{x}(0)[(4, 0, 0, \{0 :: 1\})] \qquad \{i \mapsto 1\} \mapsto \text{x}(0)[(4, 0, 0, \{0 :: 1\})]$$
$$\{i \mapsto 2\} \mapsto \text{x}(0)[(4, 0, 0, \{0 :: 1\})] \qquad \{i \mapsto 3\} \mapsto \text{x}(0)[(4, 0, 0, \{0 :: 1\})].$$

Note that an offset variable is not needed here, because the value of the rotation coincides exactly with the value that the exploded dimension i takes.

3.5 Circuit Transformations

Once a circuit is generated for the source program, the compiler has additional stages to further optimize the circuit before it generates target code.

$$\mathbf{C}_{\mathrm{e}}(\varphi_c, m) = (0, \mathbf{cipher}) \qquad \mathbf{C}_{\mathrm{e}}(\varphi_p, m) = (0, \mathbf{plain})$$

$$\frac{\mathbf{C}_{\mathrm{e}}(ce_1, m) = (v_1, \tau_1) \qquad \mathbf{C}_{\mathrm{e}}(ce_2, m) = (v_2, \tau_2)}{\mathbf{C}_{\mathrm{e}}(ce_1 \odot ce_2, m) = (v_1 + v_2 + m\mathbf{W}(\odot_{\tau_1,\tau_2}), \tau_1 \sqcup \tau_2)}$$

$$\frac{\mathbf{C}_{\mathrm{e}}(ce, mn) = (v, \tau)}{\mathbf{C}_{\mathrm{e}}(\mathbf{reduce\text{-}vec}_{\odot}(d : n, ce), m) = (v + m(n-1)\mathbf{W}(\odot_{\tau,\tau}), \tau)}$$

$$\frac{\mathbf{C}_{\mathrm{e}}(ce, m) = (v, \tau)}{\mathbf{C}_{\mathrm{e}}(\mathbf{rot}(oe, ce), m) = (v + m\mathbf{W}(\mathrm{rot}_{\tau}), \tau)}$$

$$\frac{\mathbf{C}_{\mathrm{e}}(ce, \Pi_i\ n_i) = (v, \tau)}{\mathbf{C}_{\mathrm{s}}(\mathbf{let}\ a : [d_1 : n_1, \ldots, d_n : n_n] = ce) = v} \qquad \frac{\mathbf{C}_{\mathrm{s}}(cs_1) = v_1 \qquad \mathbf{C}_{\mathrm{s}}(cs_2) = v_2}{\mathbf{C}_{\mathrm{s}}(cs_1; cs_2) = v_1 + v_2}$$

Figure 3.19: Circuit cost function.

3.5.1 Circuit Optimization

The scheduling stage of the compiler can find schedules with data layouts that result in efficient HE programs. However, there are optimizations leveraging the algebraic properties of HE operations that are missed by scheduling. The circuit optimization stage uses these algebraic properties to rewrite the circuit into an equivalent but more efficient form. The compiler performs efficient term rewriting through equality saturation [98, 107], applying rewrites to an e-graph data structure that compactly represents many equivalent circuits.

Figure 3.18 contains some identities that hold for circuit expressions. Because homomorphic addition and multiplication operate element-wise, one can view HE programs algebraically as product rings; thus the usual ring properties hold. Circuit identities also express properties of rotations and reductions. For example, rotation dis-

tributes over addition and multiplication: adding or multiplying vectors and then rotating yields the same result as rotating the vectors individually first and then adding or multiplying. Provided that the rotation amount oe does not depend on the value of the dimension variable that is being reduced (i.e. the variable is not in dim-vars(oe)) rotating vectors individually by oe and then reducing them together is the same as reducing the vectors first and then rotating the result.

Computing cost. Extraction of efficient circuits from the e-graph is guided by the cost function defined in Figure 3.19. Note that this is the same cost function that guides the search for efficient schedules during the scheduling stage. The function $\mathbf{C}_e$ takes an expression ce and its multiplicity m and returns the cost v of the expression as well as its type τ, which could either be **plain** or **cipher**. Types are ordered such that **plain** $\sqsubseteq$ **cipher**; the type for binary operations is computed from the join of its operand types according to this ordering. The cost function is parameterized by a customizable function **W** that weights operations according to their type. For example, **W** might give greater cost to operations between ciphertexts than to operations between plaintexts. The cost function also adds costs for other features of a circuit, such as the number of input vectors required (which can be computed from the circuit registry) and the multiplication depth of circuits, which is an important proxy metric for ciphertext noise that should be minimized to avoid needing using costlier encryption parameters [31].

3.5.2 Plaintext Hoisting

Not all data in an HE program are ciphertexts; instead some data such as constants and server inputs are plaintexts. Because plaintext values are known by the server,

$$\boxed{cs \rightsquigarrow cs} \qquad \boxed{\overline{d_i : n_i} \dashv ce : \tau \rightsquigarrow (cs, ce)}$$

$$\overline{d_i : n_i} \dashv \varphi_p : \textbf{plain} \rightsquigarrow (\textbf{skip}, \varphi_p) \qquad \overline{d_i : n_i} \dashv z : \textbf{plain} \rightsquigarrow (\textbf{skip}, z)$$

$$\overline{d_i : n_i} \dashv \varphi_c : \textbf{cipher} \rightsquigarrow (\textbf{skip}, \varphi_c)$$

$$\frac{\overline{d_i : n_i} \dashv ce_1 : \tau \rightsquigarrow (cs_1, ce_1') \qquad \overline{d_i : n_i} \dashv ce_2 : \tau \rightsquigarrow (cs_2, ce_2')}{\overline{d_i : n_i} \dashv ce_1 \odot ce_2 : \tau \rightsquigarrow (cs_1; cs_2, ce_1' \odot ce_2')}$$

$$\frac{\begin{array}{c}\overline{d_i : n_i} \dashv ce_1 : \textbf{plain} \rightsquigarrow (\textbf{skip}, ce_1) \\ \overline{d_i : n_i} \dashv ce_2 : \textbf{cipher} \rightsquigarrow (cs_2, ce_2') \qquad a, \varphi_p \text{ fresh}\end{array}}{\overline{d_i : n_i} \dashv ce_1 \odot ce_2 : \textbf{cipher} \rightsquigarrow \left(cs_2; \textbf{let } a : [\overline{d_i : n_i}] = ce_1, \varphi_p \odot ce_2'\right)}$$

$$\frac{\overline{d_i : n_i}, d : n \dashv ce : \tau \rightsquigarrow (cs, ce')}{\overline{d_i : n_i} \dashv \textbf{rot}(oe, ce) : \tau \rightsquigarrow (cs, \textbf{rot}(oe, ce'))}$$

$$\frac{\overline{d_i : n_i}, d : n \dashv ce : \tau \rightsquigarrow (cs, ce')}{\overline{d_i : n_i} \dashv \textbf{reduce-vec}_{\odot}(d : n, ce) : \tau \rightsquigarrow (cs, \textbf{reduce-vec}_{\odot}(d : n, ce'))}$$

$$\frac{\overline{d_i : n_i} \dashv ce : \tau \rightsquigarrow (cs_{\text{hoisted}}, ce')}{\textbf{let } a : [\overline{d_i : n_i}] = ce \rightsquigarrow cs_{\text{hoisted}}; \textbf{let } a : [\overline{d_i : n_i}] = ce'} \qquad \frac{cs_1 \rightsquigarrow cs_1' \qquad cs_2 \rightsquigarrow cs_2'}{cs_1; cs_2 \rightsquigarrow cs_1'; cs_2'}$$

Figure 3.20: Rules for plaintext hoisting.

operations between such values can be executed natively, which is more efficient than execution under HE. The plaintext hoisting stage finds circuit components that can be hoisted out and executed natively.

The compiler performs plaintext hoisting by finding maximal circuit subexpressions that perform computations only on plaintexts. Once a candidate subexpression is found, the compiler creates a let statement with the subexpression as its body. In the original circuit, the subexpression is replaced with a plaintext variable; in the circuit registry this variable is mapped to vectors that reference the output of the created let

statement.

The rules defining the plaintext hoisting pass are in Figure 3.20. The hoisting judgment for statements has the form $cs \rightsquigarrow cs'$, where input cs is transformed into cs'. Meanwhile the hoisting judgment for expressions has the form $\overline{d_i : n_i} \dashv ce : \tau \rightsquigarrow (cs, ce')$, where ce is an input circuit expression with type τ parameterized by dimensions $\overline{d_i : n_i}$, which results in a sequence of hoisted expressions cs and new transformed expression ce'. The most important rule is for $ce_1 \odot ce_2$ where ce_1 and ce_2 respectively have types **plain** and **cipher** (or vice versa); in this case, ce_1 is a maximal plaintext subexpression that can be computed natively, so it is hoisted out into a new circuit statement bound to array name a, and it is replaced with plaintext variable φ_p in the transformed expression. The circuit registry is then updated so that φ_p points to values of the new array a. The rule for let statements hoists subexpressions cs from the body expression ce and then prepends these to a transformed let statement with a new body expression ce'.

Note the hoisting defined by the rules are over-eager: e.g., a plaintext variable added to a ciphertext variable is hoisted out. In this case, the plaintext hoisting pass in the compiler deviates from the rules and will not hoist the plaintext variable into its own let statement. In practice, the compiler only hoists "complex" plaintext expressions (i.e. expressions that are not values).

3.6 Circuit Lowering

The circuit representation facilitates optimizations but is hard to translate into target code. The circuit lowering stage takes a circuit program as input and generates a *loop-nest* program that closely resembles target code. Back ends then only need to translate

$$
\begin{array}{llll}
\text{Instruction ID } i & \text{Array name } a & \text{Dimension name } d & \text{Vector } v \\
\text{Value type} & \tau_v & ::= & \textbf{native}\ (\textbf{N}) \mid \textbf{plain}\ (\textbf{P}) \mid \textbf{cipher}\ (\textbf{C}) \\
\text{Instruction type} & \tau_i & ::= & \textbf{native}\ (\textbf{N}) \mid \textbf{cipher-plain}\ (\textbf{CP}) \mid \textbf{cipher}\ (\textbf{C}) \\
\text{Array reference} & \rho_a & ::= & a \mid \rho_a[d] \\
\text{Reference} & \rho & ::= & i \mid \rho_a \\
\text{Constructor} & c & ::= & \text{Array}_{\tau_v}(\overline{n}) \mid \text{Const}(n) \mid \text{Mask}(\overline{(n,n,n)}) \mid \text{Vec}(v) \\
\text{Expression} & le & ::= & z \mid le \odot le \mid d \mid \rho \mid c \mid \textbf{encode}(\rho_a) \\
\text{Statement} & ls & ::= & \textbf{skip} \mid ls; ls \mid i \leftarrow \odot_{\tau_i}(\rho,\rho) \mid i \leftarrow \text{rot}_{\tau_i}(le,\rho) \\
 & & \mid & \rho_a := le \mid \textbf{for}\ d\ \textbf{in range}(n)\ \{\ ls\ \}
\end{array}
$$

Figure 3.21: Abstract syntax for loop-nest programs.

loop-nest programs to target code to add compiler support for HE libraries.

Figure 3.21 defines the abstract syntax of loop-nest programs. Programs manipulate arrays of vectors, which can come in three different value types. Native vectors represent data in the "native" machine representation; they cannot be used in HE computations. Plaintext vectors are encoded as HE plaintexts and can be used in HE computations. Ciphertext vectors are encrypted data from the client. Computations are represented as sequences of instructions, which are tagged with an instruction type that represents the types of their operands. Statements include instructions, assignments to arrays, and for loops. Server inputs are first declared as native vectors, and then explicitly encoded into plaintexts using the encode expression. Explicit representation of encoding allows the compiler to generate code to encode the results of computations over native vectors.

Circuit lowering translates a circuit statement **let** $a : [d_1 : n_1, \ldots, d_n : n_n] = ce$ by first generating a sequence of prelude statements that fill arrays with registry values for offset, ciphertext, and plaintext variables used in ce. The translation for the statement itself consists of a nest of n for-loops, one for each of the dimension variables d_i. The body of the loop nest is the translation for ce.

Translation of most expression forms are straightforward. Literals are replaced with references to plaintext vectors that contain the literal value in all slots. Operations and rotations are translated as instructions. The expression form **reduce-vec**$_{\odot}(d : n, ce)$ is translated by declaring an array a and a new loop that iterates over d. The body of the loop contains the translation for ce and its output is stored in the newly declared array a. After the loop, a sequence of instructions then computes the reduction as a balanced tree of operations; this is particularly important to minimize multiplication depth.[3]

Finally, a value numbering analysis over circuits prevents redundant computations in the translation to loop-nest instructions. For example, in Figure 3.6 the difference between a test point and the client-provided point is only computed once; the result is then multiplied with itself to compute the squared difference.

3.7 Implementation

We have implemented a prototype version of the Viaduct-HE compiler in about 13k LoC of Rust. The compiler uses the egg [107] equality saturation library for the circuit optimization stage. We configure egg to use the LP extractor, which lowers e-graph extraction as an integer linear program.[4] The compiler's cost estimator is tuned to reflect the relative latencies of operations and to give lower cost to plaintext-plaintext operations than ciphertext-ciphertext or ciphertext-plaintext operations (which must be executed in HE), driving the optimization stage toward circuits with plaintext hoistable components.

[3] When reducing with addition, where noise growth is not a concern, the loop instead accumulates values directly into a at the end of each iteration.

[4] The default LP extractor implementation of the egg library uses the COIN-OR CBC solver [27].

Benchmark	Vector Size	Configuration Exec Time (s)			
		baseline	e1-o0	e2-o0	e2-o1
conv-simo	4096	62.21	0.10	—	**0.09**
conv-siso	4096	15.58	0.04	—	**0.03**
distance	2048	0.54	0.37	**0.17**	—
double-matmul	4096	74.84	**0.07**	—	—
retrieval-256	8192	120.12	**0.70**	—	—
retrieval-1024	8192	585.08	1.92	**1.01**	—
set-union-16	8192	93.98	**1.01**	—	—
set-union-128	16384	>3600	**11.65**	—	—

Table 3.1: Execution time for benchmark configurations, in seconds.

Benchmark	Scheduling (s)		Circuit Opt (s)
	e1	e2	o1
conv-simo	9.43	100.03	0.003
conv-siso	1.27	14.09	0.08
distance	0.04	6.28	6.42
double-matmul	0.64	5.84	42.47
retrieval-256	0.05	0.80	170.45
retrieval-1024	0.56	16.99	5.52
set-union-16	0.06	1.66	3.60
set-union-128	7.59	663.23	9.71

Table 3.2: Compilation time for benchmark configurations, in seconds.

We have implemented a back end that targets the BFV [40] scheme implementation of the SEAL homomorphic encryption library [25]. The compiler generates Python code that calls into SEAL using the PySEAL [99] library. The back end consists of about 1k LoC of Rust and an additional 500 lines of Python. It performs a use analysis to determine when memory-efficient in-place versions of SEAL operations can be used. We use the numpy library [54] to pack arrays into vectors.

3.8 Evaluation

To evaluate Viaduct-HE, we ran experiments to determine the efficiency of vectorized HE programs generated by the compiler and to determine whether its compilation process is scalable. We used benchmarks that are either common in the literature or have been adapted from prior work. Our benchmarks are larger than those used to evaluate Porcupine [31] and Coyote [74].

Experimental setup. We ran experiments on a Dell OptiPlex 7050 machine with an 8-core Intel Core i7 7th Gen CPU and 32 GB of RAM. All numbers reported are averaged over 5 trials, with relative standard error below 8 percent.[5] We use the following programs as benchmarks:

- **conv**. A convolution over a 1-channel 32x32 client-provided image with a server-provided filter of size 3 and stride 1. The **conv-siso** variant (**s**ingle-**i**nput, **s**ingle-**o**utput) applies a single filter to the image, while the **conv-simo** variant (**s**ingle-**i**nput, **m**ultiple-**o**utput) applies 4 filters to the image.
- **distance-64**. The distance program from §3.2, but points have 64 dimensions and there are 64 test points.
- **double-matmul**. Given 16x16 matrices A_1, A_2, and B, computes $A_2 \times (A_1 \times B)$.
- **retrieval**. A private information retrieval example where the user queries a key-value store. The **retrieval-256** variant has 256 key-value pairs and 8 bit keys, while **retrieval-1024** has 1024 pairs and 10 bit keys.

[5] With the exception for the execution time reported for circuit optimization; there relative standard error is below 25 percent. The higher error is from the extractor calling into an external LP solver.

- **set-union** (from Viand et al. [104]). An aggregation from two key-value stores A and B. The program sums all the values in A and the values in B that do not share a key with some value in A. In the **set-union-16** variant A and B each have 16 key-value pairs and 4 bit keys, while in **set-union-128** A and B each have 128 key-value pairs and 7 bit keys.

We compiled these programs with various target vector sizes, shown in Table 3.1.[6] The source code and compiled programs for all benchmarks are in the supplementary materials.

3.8.1 Efficiency of Compiled Programs

To determine whether the Viaduct-HE compiler can generate efficient vectorized HE programs, we compared compiled benchmarks against baseline HE implementations using simple vectorization schedules. These baselines do not match the efficiency of expert-written implementations, but they illustrate the importance of vectorization schedules in the performance of HE programs. The baseline implementations are as follows:

- For **conv-siso** and **conv-simo**, each vector contains all the input pixels used to compute the value of a single output pixel.
- For **distance-64** the baseline implementation is the row-wise layout from Figure 3.10.
- For **double-matmul** the input matrices A_1 and B for the first multiplication are stored in vectors column- or row-wise to allow a single multiplication and then

[6] The reported vector size in Table 3.1 is half of the polynomial modulus degree parameter N, since in BFV vector slots are arranged as a $2 \times N/2$ matrix such that rotation cyclically shifts elements within rows.

rotate-and-reduce to compute a single output entry. This output layout forces A_2 to be stored as one matrix entry per vector.

- For **retrieval** and **set-union**, keys and values are stored in individual vectors.

We compared baseline implementations against implementations generated with different configurations of the Viaduct-HE compiler. For scheduling and circuit optimization, we test two configurations each: **e1** schedules for one epoch, such that the tiling transformer is disabled; **e2** schedules for two epochs; **o0** disables circuit optimization; **o1** runs circuit optimization such that equality saturation stops after either a timeout of 60 seconds or an e-graph size limit of 500 e-nodes. We did not find any optimization improvements in further increasing these limits. We use the configuration combinations **e1-o0**, **e2-o0**, and **e2-o1** in experiments.

Table 3.1 shows the results the average execution time of each benchmark under different configurations. We timed out the execution of the **set-union-128** baseline after 1 hour. For all benchmarks, Viaduct-HE implementations run faster than the baselines, with speedups ranging from 50 percent (1.45x for **distance-64** with configuration **e1-o0**) to several orders of magnitude (over 1000x for **double-matmul**). The bulk of the speedups come from the scheduling stage: only the **conv** variants show performance differences between **o0** and **o1**, since in most benchmarks circuit optimization generates the same initial circuit. We believe this is because the compiler already uses domain-specific techniques like rotate-and-reduce to generate efficient circuits before optimization, making it hard to improve on the initial circuit. Also note that most benchmarks found the optimal schedule after 1 epoch; only **distance** and **retrieval-1024** have more efficient schedules in configuration **e2** compared to **e1**.

The implementations generated by Viaduct-HE make efficient use of the SIMD capabilities of HE with sophisticated layouts. For **distance-64**, the **e1-o0** configuration

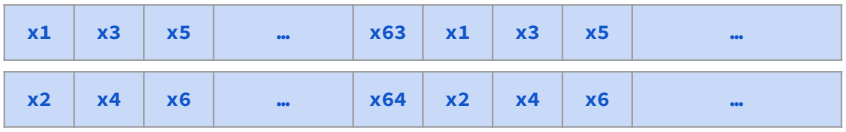

Figure 3.22: Layout for client point in **e2-o0** implementation of **distance-64**.

generates the diagonal layout from Figure 3.11, which reduces the necessary amount of rotations and additions compared to the row-wise baseline layout. The **e2-o0** configuration generates an even more efficient layout by using all 2048 vector slots available: the even and odd coordinates of the client point are packed in separate vectors and each coordinate is repeated 64 times, allowing the squared difference of each even (resp. odd) coordinate with the corresponding even (resp. odd) coordinate of each test point to be computed simultaneously. Figure 3.22 shows the layout for the client point.

Meanwhile, for **retrieval-256** the compiler generates a layout where the entire key array and the query are each stored in single vectors. Each bit of the query is repeated 256 times, allowing the equality computation with the corresponding bit of each key to be computed all at once. For **retrieval-1024**, a similar layout to **retrieval-256** is not possible because there are too many keys to store in a single vector. Instead, the **e1** configuration explodes the key array bit-wise: each bit of a key is stored in a separate vector, and the corresponding bits of all 1024 keys are packed in the same vector. The **e2** configuration, as in **distance-64**, stores the even and odd bits of keys in separate vectors to use more of the available 8192 vector slots, making it even more efficient.

3.8.2 Comparison with Expert-written HE Programs

The HE programs generated by the Viaduct-HE compiler are not only dramatically more efficient than the baseline implementations, they also sometimes match or even improve upon expert-written implementations found in the literature.

The **conv-simo** implementation generated by Viaduct-HE is basically the "packed" convolution kernel defined in Gazelle [58]: both store the image in a single ciphertext, while the values of all 4 filters at a single position are packed in a single plaintext. The image ciphertext is then rotated to align with the filter ciphertexts; since the filter size is 3x3, 9 rotated image ciphertexts and 9 filter plaintexts are multiplied together, and then summed. This computes the convolution for all output pixels at once. The **conv-siso** implementation is similar, but instead packs *columns* of the filter into a plaintext instead of single values.

The **o1** configurations of **conv-siso** and **conv-simo** use algebraic properties of circuits to optimize the implementation further. Given image ciphertext c, mask m, and plaintext filter f, instead of computing $(c \times m) \times f$ in HE as two ciphertext-plaintext operations, circuit optimization rewrites the computation as $c \times (m \times f)$, allowing $m \times f$ to be hoisted out of HE and computed natively. This is exactly the "punctured plaintexts" technique, also from Gazelle.

For **double-matmul**, Viaduct-HE generates an implementation where each matrix is laid out in a single vector. Importantly, even though A_1 and A_2 are both left operands to multiplication, their layouts are different because the layout of A_2 must account for the output layout of $A_1 \times B$. The generated implementation is similar to the expert implementation found in Dathathri et al. [33, Figure 1], but avoids a "clean-and-fill" operation required to derive an empty dimension from a reduced vectorized

dimension. The Viaduct-HE implementation avoids this operation by moving the reduced vectorized dimension as the outermost dimension in the vector, thus making it a reduced repeated dimension in the output layout of the first multiplication. The expert implementation takes 0.06 seconds compared to 0.04 seconds taken by the Viaduct-HE implementation, a 1.5x speedup.

Finally, **set-union-128** is originally a benchmark for the HECO compiler [104]. The program computes a mask that zeroes out elements of B with keys that are in A, and then adds the sum of values in A with the sum of masked values in B. The implementation generated by the HECO compiler is over 40x slower than an expert-written solution: as in the **e1** configuration of **retrieval-1024**, it packs each bit of a key in separate vector, and the corresponding bits of all 128 keys are packed in the same vector. The bits are repeated within each vector such that the computation of masks for all pairs of keys in A and B can be done simultaneously. The Viaduct-HE compiler generates exactly this expert solution.

3.8.3 Scalability of Compilation

To determine whether the Viaduct-HE compilation process is scalable, we measured the compilation time for each benchmark, with the same scheduling and optimization configurations from RQ1. Table 3.2 shows the compilation times for the benchmarks. The two main bottlenecks for compilation are the scheduling and circuit optimization stages, with their sum constituting almost all compilation time.

With 1 epoch, scheduling at most takes 10 seconds; however, with 2 epochs scheduling takes up to 11 minutes (**set-union-128**). This is because the tiling transformer vastly increases the search space, as it finds many different ways to split dimensions.

We find that scheduling is mainly hampered by the fact that circuit generation must be attempted for every visited schedule, as it is currently the only way to determine whether a schedule is valid. In particular, circuit generation is greatly slowed down by array materialization, as in many schedules (especially those with many exploded dimensions), the default array materializer generates thousands of vectors and then tries to derive these vectors from one another, so that scheduler has an accurate count of features such as the number of input vectors and rotations. Speeding up scheduling by estimating such features without array materialization is an interesting research direction.

Meanwhile, circuit optimization time is completely dominated by extraction. In all compilations, equality saturation stops in less than a second, but extraction takes longer (almost 3 minutes for **retrieval-256**) because the LP extractor must solve an integer linear program.

3.9 Related Work

Vectorized HE for Specific Applications. There is a large literature on developing efficient vectorized HE implementations of specific applications, particularly for machine learning. Some work such as Cryptonets [47], Gazelle [58], LoLa [17], and HyPHEN [62] develop efficient vectorized kernels for neural network inference. Other work such as SEALion [101] and nGraph-HE [12, 11] provide domain-specific compilers for neural networks. CHET [33] automatically selects from a fixed set of data layouts for neural network inference kernels. HeLayers [3] is similar to CHET in automating layout selection, but can also search for efficient tiling sizes for kernels, akin to the tiling transformer in Viaduct-HE. COPSE [73] develops a vectorized implemen-

tation of decision forest evaluation.

Compilers for HE. The programmability challenges of HE have inspired much recent work on HE compilers [103]. HE compilers face similar challenges as compilers for multi-party computation [55, 2, 89, 18, 24], such as lowering programs to a circuit representation. At the same time, HE has unique programmability challenges that are not comparable to other domains. Some HE compilers such as Alchemy [32], Cingulata [22], EVA [34], HECATE [68], and Ramparts [6], focus on other programmability concerns besides vectorization, such as selection of encryption parameters and scheduling "ciphertext maintenance" operations. Lobster [67] uses program synthesis and term rewriting to optimize HE circuits, but it focuses on boolean circuits and not on vectorized arithmetic circuits.

Recent work have tackled the challenge of automatically vectorizing programs for HE. Porcupine [31] proposes a synthesis-based approach to generating vectorized HE programs from an imperative source program. However, Porcupine requires the developer to provide the data layout for inputs and can only scale up to HE programs with a small number of instructions. HECO [104] attempts to solve the scalability issue by analyzing indexing operations in the source program in lieu of program synthesis, but fixes a simple layout for all programs, leaving many optimization opportunities out of reach. Coyote [74] uses search and LP to find efficient vectorizations of arithmetic circuits, balancing vectorization opportunities with data movement costs. Coyote can vectorize "irregular" programs that are out of scope for Viaduct-HE. At the same time, though it can generate layouts for HE programs, Coyote still requires user hints for "noncanonical" layouts. Also, Coyote appears to be less scalable than Viaduct-HE, as compiling a 16x16 matrix multiplication requires decomposition into 4x4 matrices that are "blocked" together.

Array-oriented Languages. In the taxonomy given by Paszke et al. [86], the Viaduct-HE source language is a "pointful" array-oriented language with explicit indexing constructs, in contrast to array "combinator" languages such as Futhark [56] and Lift [97]. The Viaduct-HE source language is thus similar in spirit to languages such as ATL [10, 70], Dex [86], and Tensor Comprehensions [102]. In particular, the separation of algorithm and schedule in Viaduct-HE is inspired by the Halide [88] language and compiler for image processing pipelines. Although the source language of Viaduct-HE is similar to Halide's—both are pointful array languages—Viaduct-HE schedules have very different concerns from Halide schedules. On one hand, Halide schedules represent choices such as what order the values of an image processing stage should be computed, and the granularity at which stage results are stored; on the other hand, Viaduct-HE schedules represent the layout of data in ciphertext and plaintext vectors.

3.10 Summary

With its array-oriented source language, the Viaduct-HE compiler can give a simple representation for vectorization schedules and find sophisticated data layouts comparable to expert HE implementations. The compiler also has representations to allow for algebraic optimizations and for easy implementation of back ends for new HE libraries. Overall, the Viaduct-HE compiler drastically lowers the programmability burden of vectorized homomorphic encryption.

CHAPTER 4

CONCLUSION

We have discussed the design and implementation of two compilers for secure computation, Viaduct and Viaduct-HE. Each of these compilers tackles important and unresolved programmability issues that impede the adoption of secure computation. The Viaduct compiler is extensible and allows programs with heteregenous security requirements to be compiled to a variety of secure computation mechanisms, addressing a limitation of all prior work. Meanwhile, the Viaduct-HE compiler can generate highly optimized, expert-level HE programs that fully use the SIMD capabilities of HE schemes. Together, they address some of the important programmability issues with using secure computation mechanisms.

4.1 Future Research Directions

While Viaduct and Viaduct-HE make significant advances to the programmability of secure computation, many fruitful research directions remain unexplored. We highlight some here.

4.1.1 Availability Labels

The label model of Viaduct captures confidentiality and integrity properties. In light of the "CIA" model of security, the lack of availability labels is an obvious lacuna. The capability to reason about availability would allow Viaduct to extend its support beyond cryptographic protocols into protocols for distributed systems in general. Availability labels in Viaduct source programs would allow developers to express requirements like

"Bob should not be able to influence whether Alice receives the result of this computation," and "A majority of Alice, Bob and Chuck must collude to prevent Dana from receiving the result." Availability labels would also allow Viaduct to approximate approximate the guarantees of fault-tolerant protocols such as quorum replication [113] and factor in availability concerns during protocol selection.

4.1.2 Trusted Hardware and Special Purpose Mechanisms

The current implementation of the Viaduct compiler focuses on purely cryptographic mechanisms for general-purpose secure computation: multi-party computation, zero-knowledge proofs, and homomorphic encryption. Extending support for trusted hardware and special purpose cryptographic mechanisms would give the Viaduct compiler a more comprehensive suite of mechanisms to target, but both pose novel challenges.

Trusted Hardware. Unlike the current protocols currently supported by Viaduct, trusted hardware cannot be given a fixed authority label: in principle, an enclave can have *any* label, since its authority is derived from the trust conferred to it by parties.[1] For example, an enclave that Alice (A) and Bob (B) fully trust would have the secret keys to decrypt their private data and to sign messages on their behalf. The authority label of the enclave would then be $A \wedge B$. The confidentiality and integrity guarantees of trusted hardware ensure that even if the machine hosting the enclave becomes corrupted, the adversary would not be able to acquire Alice or Bob's private data, and would not be able to sign messages on their behalf.

Adding support for trusted hardware in Viaduct thus complicates protocol selec-

[1] Gollamudi et al. [51] arrives at this conclusion by modeling enclaves as "computational principals" to whom parties can delegate.

tion: instead of generating as output a mapping from program components to protocols, if an enclave is used in the program then the protocol selection must now also output the required delegations for enclaves. Concretely, the delegations for an enclave are the set of secret keys for encryption and signing that the enclave has access to. Delegations can be factored into the cost model, to minimize the number of secret keys that an enclave has access to. While there is some prior work in program partitioning for trusted hardware that focuses on splitting a program running a single machine [69, 50], the approach to extend Viaduct outlined above would support *distributed* program partitioning for enclaves.

Special Purpose Mechanisms. Special purpose mechanisms for secure computation include private set intersection and oblivious RAM (ORAM). As their name implies, these mechanisms are only support specific computations. Private set intersection mechanisms, as the name implies, only support an intersection operation between two set data types, where each set contains private data by a party. Oblivious RAM mechanisms, meanwhile, allow an array stored by a server to be indexed by a client, without the server determining the access pattern used by the client (e.g., the access pattern is considered private data of the client).

Some support for such mechanisms is actually already in the Viaduct compiler: the protocol selection stage can impose syntactic restrictions that limit the kinds of operations that a protocol can be used for. For example, the protocol selection stage ensures that commitment protocol only stores data and performs no computation. Adding support for an oblivious RAM protocol to Viaduct would involve adding a similar syntactic restriction that ensures the protocol only performs array indexing operations.

Adding support for private set intersection, meanwhile, requires adding a new set

data type to the source language and operations over this data type. To avoid the proliferation of data types baked into the compiler, it would be interesting to add plug-in support for declaring new data types. A data type declartion would abstractly define a set of operations over the data type (i.e., like a Java interface). Plug-ins to other parts of the compiler would then allow compiling operations over new data types to special purpose mechanisms.

For example, to add support for a private set intersection mechanism, one can imagine the following workflow:

- Declare interface for new data type `IntSet` with a variety of operations, including an intersection operation:

```
interface IntSet {
    fun add(e: Int);
    fun contains(e: Int): Bool;
    fun intersect(other: IntSet): IntSet; // return new set
}
```

- Declare new `PSI` protocol with an appropriate authority label.[2] In the protocol selection, stage add constraints such that `PSI(A,B)` can implement the `intersect(other: IntSet)` operation given that the receiver is stored in protocol `Local(A)` and the `other` argument is a set stored in protocol `Local(B)`.
- Modify cost model to track cost of performing `intersect` in the PSI protocol.
- Modify the protocol composer such that the `PSI` protocol can communicate with `Replication(A,B)`. This allows the PSI protocol to reveal the intersection to both parties A and B.

[2] It should be the same authority as that of MPC in the same threat model (semi-honest or malicious).

- Implement the PSI back end, which involves encoding the local sets from A and B, running the PSI protocol, and then decoding the result back into an intersection set that both A and B can access locally.

4.1.3 Connecting Viaduct and Viaduct-HE

Finally, connecting Viaduct and Viaduct-HE together would address the lack of HE support in the current prototype implementation of the Viaduct compiler. This is mostly straightforward, but the main challenge is to lift programs from Viaduct's imperative source language to Viaduct-HE's array source language. As we saw in Figure 3.7, Viaduct-HE programs correspond to a loop-nest program in a traditional imperative language. If a Viaduct programs look like one of these loop-nest programs, then lifting it to a Viaduct-HE programs is simple. If Viaduct programs are more complicated, however, it is an open question how such liftings are to be done. We can take the literature on verified lifting as an inspiration: for example, Kamil et al. [59] use program synthesis techniques to automatically lift legacy Fortran programs (read: imperative programs) into Halide programs (read: array programs), which can then be optimized by the Halide compiler. One can imagine a similar technique for lifting Viaduct programs into Viaduct-HE programs.

APPENDIX A

SELECTED BENCHMARKS FOR VIADUCT

The following sections have the Viaduct source code for a select number of benchmarks and a description of the distributed programs that the compiler generates for each.

For the benchmarks used in RQ5, we also include the Kotlin code for the "bare ABY" programs with which we compared the performance of Viaduct compiled programs. The programs use the Kotlin JNI shim to ABY that the Viaduct compiler uses for its ABY back end. The Kotlin code for the most part uses the ABY API directly using the `ABYParty` class; the only code that is specific to Viaduct is `ABYCircuitBuilder`, which is a class that contains references to the arithmetic, and boolean, and Yao circuit objects used to build gates; and `Host`, which is a wrapper to the `String` class that contains the name of the current host.

Participating hosts each run a copy of the Kotlin program, so the code uses the ABY API builds the circuit for both hosts (named `alice` and `bob` by convention). In some cases the code is the same for both hosts; in other cases the code slightly differs (e.g. `alice` builds an `IN` gate while `bob` build a `DummyIN` gate), which case the code cases on which is the current host (supplied by the **`host`** parameter).

A.1 Battleship

This benchmark runs a game of battleship between Alice and Bob: each player maintains a set of ships located on a map, and then take turns attacking locations where they think an enemy ship resides. Unlike the original board game, in this version the board is one-dimensional and each ship is only 1 unit long.

To execute this program, each player provides the coordinates of their ships as input, which is stored in a private array (Lines 8–11). Then the players execute a cheating detection routine (Lines 20–30): each player reveals to the other player that their ships are not placed in the same location. In the compiled distributed program, this routine is implemented with each player sending zero-knowledge proofs to attest that the locations for each pair of their ships are not equal. A zero-knowledge proof is required here to prevent leaking the locations of the ships.

Alice and Bob then take turns attacking coordinates where they think an enemy ship is located, until one of them sinks all of the ships of the other. On Alice's turn, she takes a location to attack as input (appendix A.1) and sends this location to Bob, who then sends zero-knowledge proofs attesting whether Alice has sunk one of his battleships (Lines 46–52). Again, zero-knowledge proofs are required here to prevent leaking the locations of ships. Bob's turn is symmetric to Alice's.

```
host alice  : {A}
host bob    : {B}

// load inputs into endorsed arrays,
// so that they cannot be modified further
val aships = Array[int]{A ∧ B<-}(5);
val bships = Array[int]{B ∧ A<-}(5);
for (var i: int = 0; i < 5; i+=1) {
  aships[i] = endorse (input int from alice) from {A};
  bships[i] = endorse (input int from bob) from {B};
}

var awins: bool{A ⊓ B} = false;

// if someone put multiple battleships in the same cell,
// they automatically lose
var acheated: bool{A ⊓ B} = false;
var bcheated: bool{A ⊓ B} = false;

for (var j: int{A ⊓
   B} = 0; j < 5 ∧ !acheated ∧ !bcheated; j += 1) {
  for (var k: int{A ⊓
   B} = j + 1; k < 5 ∧ !acheated ∧ !bcheated; k += 1) {
    if (declassify (aships[j] == aships[k]) to {A ⊓ B}) {
      acheated = true;
    }

    if (declassify(bships[j] == bships[k]) to {A ⊓ B}) {
      bcheated = true;
```

```
    }
  }
}


if (!acheated ∧ !bcheated) {
  var ascore: int{A ⊓ B} = 0;
  var bscore: int{A ⊓ B} = 0;

  var playing: bool{A ⊓ B} = true;
  var aturn: bool{A ⊓ B} = true;

  // keep playing until someone sinks all the other person's
   battleships
  while (playing) {
    if (aturn) {
      val amove: int{A ⊓ B->} =
        declassify (input int from alice) to {A ⊓
   B->};
      var amove_trusted: int{A ⊓ B} = endorse amove from {A ⊓
   B->};
      var ahit: bool{A ⊓ B} = false;
      for (var aj: int{A ⊓
   B} = 0; aj < 5; aj += 1) {
        if (declassify (bships[aj] == amove_trusted) to {A ⊓
   B}) {
          ascore += 1;
          bships[aj] = 0;
          ahit = true;
        }
      }


      output ahit to alice;
      output ahit to bob;
      aturn = false;
    } else {
      var bmove: int{B ⊓ A->} =
        declassify (input int from bob) to {B ⊓
   A->};
      val bmove_trusted: int{A ⊓ B} = endorse bmove from {B ⊓
   A->};

      var bhit: bool{A ⊓ B} = false;
      for (var bj: int{A ⊓ B} = 0; bj < 5; bj += 1) {
        if (declassify (aships[bj] == bmove_trusted) to {A ⊓
   B}) {
          bscore += 1;
          aships[bj] = 0;
          bhit = true;
        }
      }

      output bhit to alice;
      output bhit to bob;
      aturn = true;
    }
```

```
    playing = ascore < 5 ∧ bscore < 5;
  }

  awins = ascore == 5;
  output awins to alice;
  output awins to bob;
} else {
  output bcheated to alice;
  output bcheated to bob;
}
```

A.2 Biometric Matching

This benchmark computes the minimum Euclidean distance of Bob's sample to some region in Alice's database, a common routine in bioinformatics. The Euclidean distance is computed by the `match` function, which takes as input two points in Alice's database (`db1`, `db2`) and Bob's sample (`s1`, `s2`) and returns the Euclidean distance between these, given as the **`out`** parameter `res`. Note that the labels for the formal parameters of `match` are upper-bounds; in the Viaduct source language, the concrete label of the arguments at a call site can be referenced in the body of a function by using the parameter name corresponding to the argument, as seen in the labels for `dist1` and `dist2` (Lines 8–9).

In the compiled implementation generated by Viaduct, Alice and Bob store their respective database and samples locally and then use an MPC protocol to compute the minimum Euclidean distance.

```
host alice: {A ∧ B<-}
host bob: {B ∧ A<-}

fun match(
  db1: int{A ∧ B<-}, db2: int{A ∧ B<-}, s1: int{B ∧ A<-}, s2: int{B
   ∧ A<-},
  res: out int{A ∧ B}
) {
  val dist1: int{db1 ∧ s1} = db1 - s1;
  val dist2: int{db2 ∧ s2} = db2 - s2;
  out res = (dist1 * dist1) + (dist2 * dist2);
}
```

```
val n: int{A ⊓ B} = 500;
val d: int{A ⊓ B} = 2;

val a_db = Array[int]{A ∧ B<-}(n * d);
val b_sample = Array[int]{B ∧ A<-}(d);

for (var i: int{A ⊓ B} = 0; i < n*d; i += 1) {
  a_db[i] = input int from alice;
}

for (var i: int{A ⊓ B} = 0; i < d; i += 1) {
  b_sample[i] = input int from bob;
}

match(a_db[0], a_db[1], b_sample[0], b_sample[1], val init_min)
  ;
var min_dist: int{A ∧ B} = init_min;

for (var i: int{A ⊓ B} = 0; i < n; i += 1) {
  match(a_db[(i*d)], a_db[(i*d)+1], b_sample[0], b_sample[1],
   val dist);

  if (dist < min_dist) {
    min_dist = dist;
  }
}

val result: int{A ⊓ B}  = declassify min_dist to {A ⊓ B};
output result to alice;
output result to bob;
```

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

```
fun match_alice(db1: Int, db2: Int): Share {
  val tmp = builder.arithCircuit.putINGate(db1.toBigInteger(),
   BITLEN, builder.role)
  val tmp1 = builder.arithCircuit.putDummyINGate(BITLEN)
  val dist1 = builder.arithCircuit.putSUBGate(tmp, tmp1)

  val tmp3 = builder.arithCircuit.putINGate(db2.toBigInteger(),
    BITLEN, builder.role)
  val tmp4 = builder.arithCircuit.putDummyINGate(BITLEN)
  val dist2 = builder.arithCircuit.putSUBGate(tmp3, tmp4)

  val tmp8 = builder.arithCircuit.putMULGate(dist1, dist1)
  val tmp11 = builder.arithCircuit.putMULGate(dist2, dist2)
  val tmp12 = builder.arithCircuit.putADDGate(tmp8, tmp11)
  return builder.yaoCircuit.putA2YGate(tmp12)
}

fun match_bob(s1: Int, s2: Int): Share {
  val tmp = builder.arithCircuit.putDummyINGate(BITLEN)
  val tmp1 = builder.arithCircuit.putINGate(s1.toBigInteger(),
```

```
    BITLEN, builder.role)
  val dist1 = builder.arithCircuit.putSUBGate(tmp, tmp1)

  val tmp3 = builder.arithCircuit.putDummyINGate(BITLEN)
  val tmp4 = builder.arithCircuit.putINGate(s2.toBigInteger(),
    BITLEN, builder.role)
  val dist2 = builder.arithCircuit.putSUBGate(tmp3, tmp4)

  val tmp8 = builder.arithCircuit.putMULGate(dist1, dist1)
  val tmp11 = builder.arithCircuit.putMULGate(dist2, dist2)
  val tmp12 = builder.arithCircuit.putADDGate(tmp8, tmp11)
  return builder.yaoCircuit.putA2YGate(tmp12)
}


fun benchLANBiomatch(host: Host, aby: ABYParty, builder:
    ABYCircuitBuilder) {
  val n = 500
  val d = 4

  when (host) {
    'alice' => {
      val a_db = Array<Int>(n * d) { 0 }
      var i = 0
      while (i < n * d) {
        a_db[i] = input.nextInt()
        i += 1
      }

      var min_dist = match_alice(a_db[0], a_db[1])
      var i_2 = 0
      while (i_2 < n) {
        val db1 = a_db[i_2 * d]
        val db2 = a_db[(i_2 * d) + 1]
        val dist = match_alice(db1, db2)
        val tmp50 = builder.yaoCircuit.putGTGate(min_dist, dist
    )
        val mux = builder.yaoCircuit.putMUXGate(dist, min_dist,
     tmp50)
        min_dist = mux
        i_2 += 1
      }

      val out = builder.yaoCircuit.putOUTGate(min_dist, Role.
    ALL)
      executeABYCircuit(aby)
      println(out.clearValue32.toInt())
    }

    'bob' => {
        val b_sample = Array<Int>(d) { 0 }
        var i = 0
        while (i < d) {
          b_sample[i] = input.nextInt()
          i += 1
        }
```

```
        var min_dist = match_bob(b_sample[0], b_sample[1])
        var i_2 = 0
        while (i_2 < n) {
          val s1 = b_sample[0]
          val s2 = b_sample[1]
          val dist = match_bob(s1, s2)
          val tmp50 = builder.yaoCircuit.putGTGate(min_dist,
   dist)
          val mux = builder.yaoCircuit.putMUXGate(dist,
   min_dist, tmp50)
          min_dist = mux
          i_2 += 1
        }

        val out = builder.yaoCircuit.putOUTGate(min_dist, Role.
   ALL)
        executeABYCircuit(aby)
        println(out.clearValue32.toInt())
    }

    else => throw ViaductInterpreterError('unknown host')
  }
}
```

A.3 Interval

This benchmarks computes the interval in which Alice and Bob's private points reside, and then checks whether Chuck's private point resides in the interval. In the compiled implementation generated by the Viaduct compiler, Alice and Bob execute an MPC protocol to compute the interval in which their points lie (Lines 23–appendix A.3). They then send the interval to Chuck, who sends either Alice or Bob a zero-knowledge proof to attest whether his point lies within the interval (appendix A.3). If Alice receives the zero-knowledge proof, she verifies and then sends the result to Bob, and then they both output the result. The case where Bob receives the zero-knowledge proof is symmetric.

```
host alice   : {A ∧ B<-}
host bob     : {B ∧ A<-}
host chuck   : {C}

// Chuck can read these public parameters,
// but doesn't need to trust them since he is not using them
val a_num_points: int{A ⊓ B ⊓ C->} = 5;
val b_num_points: int{A ⊓ B ⊓ C->} = 5;
```

```
val num_points: int{A ⊓ B ⊓ C->} = a_num_points + b_num_points;

val chuck_point: int{C ∧ (A∧B)<-} =
  endorse (input int from chuck) to {C ∧ (A∧B)<-} from {C};

val points = Array[int]{A ∧ B}(num_points);
for (var i: int{A ⊓ B ⊓ C->} = 0; i < a_num_points; i += 1) {
  points[i] = input int from alice;
}

for (var i: int{A ⊓ B ⊓ C->} = 0; i < b_num_points; i += 1) {
  points[a_num_points+i] = input int from bob;
}

var min_point: int{A ∧ B} = points[0];
var max_point: int{A ∧ B} = points[0];

for (var i: int{A ⊓ B ⊓ C->} = 1; i < num_points; i += 1) {
  min_point = min(min_point, points[i]);
  max_point = max(max_point, points[i]);
}

val min_point_public: int{A ⊓ B ⊓ C->} =
  declassify min_point to {A ⊓ B ⊓ C->};

val max_point_public: int{A ⊓ B ⊓ C->} =
  declassify max_point to {A ⊓ B ⊓ C->};

val min_point_trusted: int{A ⊓ B ⊓ C} =
  endorse min_point_public from {A ⊓ B ⊓ C->};

val max_point_trusted: int{A ⊓ B ⊓ C} =
  endorse max_point_public from {A ⊓ B ⊓ C->};

val in_interval: bool{C ∧ (A∧B)<-} =
  min_point_trusted <= chuck_point ∧ chuck_point <=
   max_point_trusted;

// Chuck doesn't need to trust this because
// it will not be part of his output
val in_interval_public: bool{A ⊓ B ⊓ C->} =
  declassify in_interval to {A ⊓ B ⊓ C};

output in_interval_public to alice;
output in_interval_public to bob;
```

A.4 k-means clustering

This benchmark runs a k-means clustering algorithm over Alice and Bob's private data points. The compiled implementation executes the algorithm in an MPC protocol (Lines 25–79). After the algorithm finishes, the coordinates of the cluster centroids are declassified to both participants (Lines 82–86).

```
host alice  : {A ∧ B<-}
host bob    : {B ∧ A<-}

val a_len: int{A ⊓ B} = 50;
val b_len: int{A ⊓ B} = 50;
val len: int{A ⊓ B} = a_len + b_len;
val dim: int{A ⊓ B} = 2;
val num_clusters: int{A ⊓ B} = 4;
val num_iter: int{A ⊓ B} = 3;

val data = Array[int]{A ∧ B}(len * dim);

// load data
for (var i: int{A ⊓ B} = 0; i < a_len * dim; i += 1) {
  data[i] = input int from alice;
}

for (var i: int{A ⊓ B} = 0; i < b_len * dim; i += 1) {
  data[(a_len*dim) + i] = input int from bob;
}

val clusters = Array[int]{A ∧ B}(num_clusters * dim);

// initialize by picking data points as centroids in a stride
val stride: int{A ⊓ B} = len / num_clusters;
for (var c: int{A ⊓ B} = 0; c < num_clusters; c += 1) {
  for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
    clusters[(c*dim)+d] = data[(stride*c*dim)+d];
  }
}

for (var iter: int{A ⊓ B} = 0; iter < num_iter; iter += 1) {
  // assign points to clusters
  val best_clusters = Array[int]{A ∧ B}(len);
  for (var i: int = 0; i < len; i += 1) {

    // initialize to first cluster
    var best_dist: int{A ∧ B} = 0;
    var best_cluster: int{A ∧ B} = 0;
    for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
      val sub: int{A ∧ B} = data[(i*dim)+d] - clusters[d];
      best_dist += sub * sub;
    }
```

```
    for (var c: int{A ⊓ B} = 1; c < num_clusters; c += 1) {
      var dist: int{A ⋀ B} = 0;
      for (var d: int{A ⊓ B}; d < dim; d += 1) {
        val sub: int{A ⋀
   B} = data[(i*dim)+d] - clusters[(c*dim)+d];
        dist += sub * sub;
      }

      best_cluster = dist < best_dist ? c : best_cluster;
    }

    best_clusters[i] = best_cluster;
  }

  // update cluster centroids
  for (var c: int{A ⊓ B} = 0; c < num_clusters; c += 1) {
    val new_centroid_sum = Array[int]{A ⋀ B}(dim);
    var num_points: int{A ⋀ B} = 0;
    for (var i: int = 0; i < len; i += 1) {
      val in_cluster: bool{A ⋀ B} = best_clusters[i] == c;

      for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
        new_centroid_sum[d] += in_cluster ? data[(i*dim)+d] :
   0;
      }

      if (in_cluster) {
        num_points += 1;
      }
    }

    for (var d: int{A ⊓ B} = 0; d < dim; d += 1) {
      clusters[(c*dim)+d] = num_points > 0 ?
        (new_centroid_sum[d] / num_points) : clusters[(c*dim)+d
   ]);
    }
  }
}


// declassify clusters
for (var h: int{A ⊓
   B} = 0; h < num_clusters * dim; h += 1) {
  val public_cluster: int{A ⊓ B} = declassify clusters[h] to {A ⊓
   B};
  output public_cluster to alice;
  output public_cluster to bob;
}
```

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

```
fun kmeans(host: Host, aby: ABYParty, builder:
   ABYCircuitBuilder) {
```

```
val a_len = 50
val b_len = 50
val len = a_len + b_len
val dim = 2
val num_clusters = 4
val num_iterations = 3

// YaoABY
val data = Array<Share?>(len * dim) { null }

when (host) {
  'alice' => {
    var i = 0
    while (i < a_len * dim) {
      val x = input.nextInt()
      data[i] = builder.yaoCircuit.putINGate(x.toBigInteger()
 , BITLEN, builder.role)
      i += 1
    }

    var i_1 = 0
    while (i_1 < b_len * dim) {
      data[(a_len * dim) + i_1] = builder.yaoCircuit.
 putDummyINGate(BITLEN)
      i_1 += 1
    }
  }

  'bob' => {
    var i = 0
    while (i < a_len * dim) {
      data[i] = builder.yaoCircuit.putDummyINGate(BITLEN)
      i += 1
    }

    var i_1 = 0
    while (i_1 < b_len * dim) {
      val x = input.nextInt()
      data[(a_len * dim) + i_1] =
        builder.yaoCircuit.putINGate(x.toBigInteger(), BITLEN
 , builder.role)
      i_1 += 1
    }
  }

  else => throw Error('unknown host')
}

// ArithABY
val clusters = Array<Share?>(num_clusters * dim) { null }
val stride = len / num_clusters

var c = 0
while (c < num_clusters) {
  var d = 0
  while (d < dim) {
    clusters[(c * dim) + d] =
```

```
      builder.arithCircuit.putY2AGate(data[(stride * c * dim)
  + d], builder.boolCircuit)
      d += 1
    }
    c += 1
  }

  var iter = 0
  while (iter < num_iterations) {
    // YaoABY
    val best_clusters = Array<Share?>(len) { null }

    // assignment phase
    var i = 0
    while (i < len) {
      var best_dist = builder.arithCircuit.putCONSGate(0.
 toBigInteger(), BITLEN)
      var best_cluster = builder.yaoCircuit.putCONSGate(0.
 toBigInteger(), BITLEN)

      // initialize point to first cluster
      var d = 0
      while (d < dim) {
        val tmp62 =
          builder.arithCircuit.putB2AGate(
              builder.boolCircuit.putY2BGate(data[(i * dim) + d
 ])
          )
        val sub = builder.arithCircuit.putSUBGate(tmp62,
 clusters[d])
        val tmp68 = builder.arithCircuit.putMULGate(sub, sub)
        best_dist = builder.arithCircuit.putADDGate(best_dist,
 tmp68)

        d += 1
      }

      // assign point to nearest cluster
      var c2 = 1
      while (c2 < num_clusters) {
        var dist = builder.arithCircuit.putCONSGate(0.
 toBigInteger(), BITLEN)
        var d2 = 0
        while (d2 < dim) {
          val tmp80 =
              builder.arithCircuit.putB2AGate(
                  builder.boolCircuit.putY2BGate(data[(i * dim)
  + d2])
              )
          val sub = builder.arithCircuit.putSUBGate(tmp80,
 clusters[(c2 * dim) + d2])
          val tmp90 = builder.arithCircuit.putMULGate(sub, sub)
          dist = builder.arithCircuit.putADDGate(dist, tmp90)
          d2 += 1
        }

        val tmp91 = builder.yaoCircuit.putA2YGate(dist)
```

```
      val tmp92 = builder.yaoCircuit.putA2YGate(best_dist)
      val tmp93 = builder.yaoCircuit.putGTGate(tmp92, tmp91)
      val tmp94 = builder.yaoCircuit.putCONSGate(c2.
toBigInteger(), BITLEN)
      val tmp96 = builder.yaoCircuit.putMUXGate(tmp94,
best_cluster, tmp93)
      best_cluster = tmp96
      c2 += 1
    }

    best_clusters[i] = best_cluster
    i += 1
  }

  // update phase
  var c3 = 0
  while (c3 < num_clusters) {
    // YaoABY
    val new_centroid_sum = Array<Share?>(dim) {
        builder.yaoCircuit.putCONSGate(0.toBigInteger(),
BITLEN)
    }
    var num_points = builder.yaoCircuit.putCONSGate(0.
toBigInteger(), BITLEN)
    var i2 = 0
    while (i2 < len) {
      val tmp108 = builder.yaoCircuit.putCONSGate(c3.
toBigInteger(), BITLEN)
      val in_cluster = builder.yaoCircuit.putEQGate(
best_clusters[i2], tmp108)
      var d3 = 0
      while (d3 < dim) {
        val tmp121 =
          builder.yaoCircuit.putMUXGate(
              data[(i2 * dim) + d3],
              builder.yaoCircuit.putCONSGate(0.toBigInteger()
, BITLEN),
              in_cluster
          )

        new_centroid_sum[d3] = builder.yaoCircuit.putADDGate(
new_centroid_sum[d3], tmp121)
        d3 += 1
      }

      val op =
          builder.yaoCircuit.putADDGate(
              num_points,
              builder.yaoCircuit.putCONSGate(1.toBigInteger()
, BITLEN)
          )
      val mux = builder.yaoCircuit.putMUXGate(op, num_points,
 in_cluster)
      num_points = mux
      i2 += 1
    }
```

```
      var d4 = 0
      while (d4 < dim) {
        val tmp132 =
            builder.yaoCircuit.putGTGate(
                num_points,
                builder.yaoCircuit.putCONSGate(0.toBigInteger()
  , BITLEN)
            )

        val tmp136 =
          Aby.putInt32DIVGate(builder.yaoCircuit, num_points,
  new_centroid_sum[d4])

        val tmp142 =
          builder.yaoCircuit.putA2YGate(clusters[(c3 * dim) +
  d4])

        clusters[(c3 * dim) + d4] =
          builder.arithCircuit.putB2AGate(
              builder.boolCircuit.putY2BGate(
                  builder.yaoCircuit.putMUXGate(tmp136, tmp142,
   tmp132)
              )
          )

        d4 += 1
      }

      c3 += 1
    }

    iter += 1
  }

  var h = 0
  var out_gates = Array<Share?>(num_clusters * dim) {
    builder.arithCircuit.putCONSGate(0.toBigInteger(), BITLEN)
  }
  while (h < num_clusters * dim) {
    out_gates[h] = builder.arithCircuit.putOUTGate(clusters[h],
    Role.ALL)
    h += 1
  }

  aby.execCircuit()

  var i = 0
  while (i < num_clusters * dim) {
    println(out_gates[i]!!.clearValue32.toInt())
    i += 1
  }
}
```

A.5 Rock–Paper–Scissors

Alice and Bob play a game of rock–paper–scissors.

In the compiled implementation, Alice and Bob input their moves ahead of time and send each other commitments to their moves (Lines 10–13). Then the turns of the game are played by opening the commitments to Alice and Bob's moves for that turn and awarding the winning player a point (Lines 19–53). If a player's input is invalid, the other player is awarded a point. At the end of the game, the winner is determined and sent as output to the players (Lines 56–58).

```
host alice : {A}
host bob   : {B}

val num_turns: int{A ⊓ B} = 3;
var a_score: int{A ⊓ B} = 0;
var b_score: int{A ⊓ B} = 0;
val a_moves = Array[int]{A ∧ B<-}(num_turns);
val b_moves = Array[int]{B ∧ A<-}(num_turns);

for (var i: int{A ⊓ B} = 0; i < num_turns; i += 1) {
  a_moves[i] = endorse (input int from alice) from {A};
  b_moves[i] = endorse (input int from bob) from {B};
}

for (var turn: int{A ⊓ B} = 0; turn < num_turns; turn += 1) {
  val a_move: int{A ∧ B<-} = a_moves[turn];
  val b_move: int{B ∧ A<-} = b_moves[turn];

  val a_move_public: int{A ⊓ B} = declassify a_move to {A ⊓ B};
  val b_move_public: int{A ⊓ B} = declassify b_move to {A ⊓ B};

  // 1 = rock; 2 = paper; 3 = scissors;
  val a_valid: bool{A ⊓
   B} = 1 <= a_move_public ∧ a_move_public <= 3;
  val b_valid: bool{A ⊓
   B} = 1 <= b_move_public ∧ b_move_public <= 3;

  // alice cheats
  if (!a_valid ∧ b_valid) {
    b_score += 1;
  }

  // bob cheats
  if (a_valid ∧ !b_valid) {
    a_score += 1;
  }
```

```
  // neither cheat
  if (a_valid ∧ b_valid) {
    if (a_move_public < b_move_public ∧ b_move_public < 3) {
      b_score += 1;
    }

    if (b_move_public < a_move_public ∧ a_move_public < 3) {
      a_score += 1;
    }

    if (a_move_public == 1 ∧ b_move_public == 3) {
      a_score += 1;
    }

    if (b_move_public == 1 ∧ a_move_public == 3) {
      b_score += 1;
    }
  }
}

val a_wins: bool{A ⊓ B} = a_score > b_score;
output a_wins to alice;
output a_wins to bob;
```

A.6 Two-Round Bidding

Alice and Bob participate in auctions for n items. The auction occurs in two rounds. First, Alice and Bob place bids on each item. The first-round winner for each item is then revealed. Next, Alice and Bob place a second bid on each item. The overall winner for an item is the person who places the highest average bid between the two rounds.

To prevent leaking the actual values of their bids, which is supposed to be kept private, Alice and Bob execute an MPC protocol to perform the comparisons between their bids (appendix A.6 and appendix A.6). The rest of the program can be executed in cleartext.

```
host alice: {A ∧ B<-}
host bob:   {B ∧ A<-}

val n: int{A ⊓ B} = 500; // number of items to bid
val abids1 = Array[int]{A ∧ B<-}(n);
val abids2 = Array[int]{A ∧ B<-}(n);
```

```
val bbids1 = Array[int]{B ∧ A<-}(n);
val bbids2 = Array[int]{B ∧ A<-}(n);

// round 1
for (var i: int{A ⊓ B} = 0; i < n; i += 1) {
  abids1[i] = input int from alice;
  bbids1[i] = input int from bob;
}

// reveal first-round winners
for (var i: int{A ⊓ B} = 0; i < n; i += 1) {
  val winner: bool = declassify abids1[i] < bbids1[i] to {A ⊓
   B};
  output winner to alice;
  output winner to bob;
}

// round 2
for (var i: int{A ⊓ B} = 0; i < n; i += 1) {
  abids2[i] = input int from alice;
  bbids2[i] = input int from bob;
}

// reveal overall winners
for (var i: int{A ⊓ B} = 0; i < n; i += 1) {
  val abid: int{A ∧ B<-} = (abids1[i] + abids2[i]) / 2;
  val bbid: int{B ∧ A<-} = (bbids1[i] + bbids2[i]) / 2;
  val winner: bool{A ⊓ B} = declassify abid < bbid to {A ⊓
   B};
  output winner to alice;
  output winner to bob;
}
```

The program is compiled to one semantically equivalent to the Kotlin program below that uses ABY directly.

```
fun twoRoundBidding(host: Host, aby: ABYParty, builder:
   ABYCircuitBuilder) {
  val n = 500
  when (host) {
    'alice' => {
      val abids1 = Array<Int>(n) { 0 }
      val abids2 = Array<Int>(n) { 0 }

      var i = 0
      while (i < n) {
        abids1[i] = input.nextInt()
        i += 1
      }

      var i_1 = 0
      while (i_1 < n) {
        val tmp15 =
          builder.yaoCircuit.putINGate(
            abids1[i_1].toBigInteger(), BITLEN, builder.role
```

```
        )
      val tmp17 = builder.yaoCircuit.putDummyINGate(BITLEN)
      val tmp18 = builder.yaoCircuit.putGTGate(tmp17, tmp15)
      val tmp19 = builder.yaoCircuit.putOUTGate(tmp18, Role.
ALL)

      aby.execCircuit()

      val winner = tmp19.clearValue32.toInt()

      aby.reset()

      println(winner)

      i_1 += 1
    }

    var i_2 = 0
    while (i_2 < n) {
      abids1[i_2] = input.nextInt()
      i_2 += 1
    }

    var i_3 = 0
    while (i_3 < n) {
      val abid =
        builder.yaoCircuit.putINGate(
          ((abids1[i_3] + abids2[i_3]) / 2).toBigInteger(),
          BITLEN,
          builder.role
        )
      val bbid = builder.yaoCircuit.putDummyINGate(BITLEN)
      val tmp46 = builder.yaoCircuit.putGTGate(bbid, abid)
      val tmp47 = builder.yaoCircuit.putOUTGate(tmp46, Role.
ALL)

      aby.execCircuit()

      val winner_1 = tmp47.clearValue32.toInt()

      aby.reset()

      println(winner_1)

      i_3 += 1
    }
  }

  'bob' => {
    val bbids1 = Array<Int>(n) { 0 }
    val bbids2 = Array<Int>(n) { 0 }

    var i = 0
    while (i < n) {
      bbids1[i] = input.nextInt()
      i += 1
    }
```

```
      var i_1 = 0
      while (i_1 < n) {
        val tmp15 = builder.yaoCircuit.putDummyINGate(BITLEN)
        val tmp17 =
          builder.yaoCircuit.putINGate(
            bbids1[i_1].toBigInteger(), BITLEN, builder.role
          )
        val tmp18 = builder.yaoCircuit.putGTGate(tmp17, tmp15)
        val tmp19 = builder.yaoCircuit.putOUTGate(tmp18, Role.
    ALL)

        aby.execCircuit()

        val winner = tmp19.clearValue32.toInt()

        aby.reset()

        println(winner)

        i_1 += 1
      }

      var i_2 = 0
      while (i_2 < n) {
        bbids1[i_2] = input.nextInt()
        i_2 += 1
      }

      var i_3 = 0
      while (i_3 < n) {
        val abid = builder.yaoCircuit.putDummyINGate(BITLEN)
        val bbid =
          builder.yaoCircuit.putINGate((
            (bbids1[i_3] + bbids2[i_3]) / 2).toBigInteger(),
            BITLEN,
            builder.role
          )
        val tmp46 = builder.yaoCircuit.putGTGate(bbid, abid)
        val tmp47 = builder.yaoCircuit.putOUTGate(tmp46, Role.
    ALL)

        aby.execCircuit()

        val winner_1 = tmp47.clearValue32.toInt()

        aby.reset()

        println(winner_1)

        i_3 += 1
      }
     }

     else => throw ViaductInterpreterError('unknown host')
   }
}
```

APPENDIX B

SELECTED BENCHMARKS FOR VIADUCT-HE

This section contains the source code and implementations generated by the Viaduct compiler, given in the loop-nest representation.

B.1 Source code

conv-simo

```
input img: [32,32] from client
input filter: [4,3,3] from server
for x: 30 {
  for y: 30 {
    for out: 4 {
      sum(for i: 3 {
        sum(for j: 3 {
          img[x + i][y + j] * filter[out][i][j]
        })
      })
    }
  }
}
```

conv-siso

```
input img: [32,32] from client
input filter: [3,3] from server
for x: 30 {
  for y: 30 {
    sum(for i: 3 {
      sum(for j: 3 {
        img[x + i][y + j] * filter[out][i][j]
      })
    })
  }
}
```

distance

```
input point: [64] from client
input tests: [64,64] from server
for i: 64 {
  sum(for j: 64 {
    (point[j] - tests[i][j]) * (point[j] - tests[i][j])
  })
}
```

matmul-2

```
input A1: [16,16] from server
input A2: [16,16] from server
input B: [16,16] from client
let res =
  for i: 16 {
    for j: 16 {
      sum(for k: 16 { A1[i][k] * B[k][j] })
    }
  }
in
for i: 16 {
  for j: 16 {
    sum(for k: 16 { A2[i][k] * res[k][j] })
  }
}
```

retrieval-256

```
input keys: [256,8] from client
input values: [256] from client
input query: [8] from client
let mask =
  for i: 256 {
    product(for j: 8 {
      1 - ((query[j] - keys[i][j]) * (query[j] - keys[i][j]))
    })
  }
in
sum(values * mask)
```

retrieval-1024

```
input keys: [1024,10] from client
input values: [1024] from client
input query: [10] from client
let mask =
  for i: 1024 {
    product(for j: 10 {
      1 - ((query[j] - keys[i][j]) * (query[j] - keys[i][j]))
    })
  }
in
sum(values * mask)
```

set-union-16

```
input a_id: [16, 4] from client
input a_data: [16] from client
input b_id: [16, 4] from client
input b_data: [16] from client
let a_sum = sum(a_data) in
let b_sum =
```

```
  sum(for j: 16 {
    b_data[j] *
    product(for i: 16 {
      1 -
      product(for k: 4 {
        1 - ((a_id[i][k] - b_id[j][k]) * (a_id[i][k] - b_id[j][
   k]))
      })
    })
  })
in
a_sum + b_sum
```

set-union-128

```
input a_id: [128, 7] from client
input a_data: [128] from client
input b_id: [128, 7] from client
input b_data: [128] from client
let a_sum = sum(a_data) in
let b_sum =
  sum(for j: 128 {
    b_data[j] *
    product(for i: 128 {
      1 -
      product(for k: 7 {
        1 - ((a_id[i][k] - b_id[j][k]) * (a_id[i][k] - b_id[j][
   k]))
      })
    })
  })
in
a_sum + b_sum
```

B.2 Implementations

conv-simo e1-o0

```
val v_img_1: C = vector(img(0, 0)[(32, 0, 0 {1 :: 1}), (4, 0,
   0, {}), (32, 0, 0 {0 :: 1})])
val v_filter_1: N = vector(filter(0, 2, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_2: N = vector(filter(0, 2, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_3: N = vector(filter(0, 0, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_4: N = vector(filter(0, 1, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_5: N = vector(filter(0, 2, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
```

```
val v_filter_6: N = vector(filter(0, 0, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_7: N = vector(filter(0, 0, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_8: N = vector(filter(0, 1, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_9: N = vector(filter(0, 1, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val mask_1: N = mask([(32, 0, 30), (4, 0, 3), (32, 0, 30)])
encode(v_filter_3)
encode(v_filter_4)
encode(v_filter_1)
encode(v_filter_9)
encode(v_filter_5)
encode(v_filter_7)
encode(v_filter_8)
encode(v_filter_2)
encode(v_filter_6)
encode(mask_1)
var pt2: P[3][3] = 0
pt2[0][0] = v_filter_7
pt2[0][1] = v_filter_8
pt2[0][2] = v_filter_5
pt2[1][0] = v_filter_3
pt2[1][1] = v_filter_9
pt2[1][2] = v_filter_2
pt2[2][0] = v_filter_6
pt2[2][1] = v_filter_4
pt2[2][2] = v_filter_1
var __out: C = 0
var __reduce_2: C = 0
for i2 in range(3) {
    var __reduce_1: C = 0
    for i7 in range(3) {
        instr1 = rot(CC, ((0 + (-128 * i7)) + (-1 * i2)),
   v_img_1)
        instr3 = mul(CP, instr1, mask_1)
        instr5 = mul(CP, instr3, pt2[i7][i2])
        instr6 = add(CC, __reduce_1, instr5)
        __reduce_1 = instr6
    }
    instr8 = add(CC, __reduce_2, __reduce_1)
    __reduce_2 = instr8
}
__out = __reduce_2
```

conv-simo e2-o1

```
val v_img_1: C = vector(img(0, 0)[(32, 0, 0 {0 :: 1}), (4, 0,
   0, {}), (32, 0, 0 {1 :: 1})])
val v_filter_1: N = vector(filter(0, 0, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_2: N = vector(filter(0, 2, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_3: N = vector(filter(0, 1, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
```

```
val v_filter_4: N = vector(filter(0, 0, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_5: N = vector(filter(0, 2, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_6: N = vector(filter(0, 0, 1)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_7: N = vector(filter(0, 1, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_8: N = vector(filter(0, 2, 2)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val v_filter_9: N = vector(filter(0, 1, 0)[(30, 0, 2, {}), (4,
   0, 0 {0 :: 1}), (30, 0, 2, {})])
val mask_1: N = mask([(32, 0, 30), (4, 0, 3), (32, 0, 30)])
var pt2: P[3][3] = 0
pt2[0][0] = v_filter_4
pt2[0][1] = v_filter_9
pt2[0][2] = v_filter_2
pt2[1][0] = v_filter_6
pt2[1][1] = v_filter_3
pt2[1][2] = v_filter_5
pt2[2][0] = v_filter_1
pt2[2][1] = v_filter_7
pt2[2][2] = v_filter_8
var __partial_1: N[3][3] = 0
for i2 in range(3) {
    for i6 in range(3) {
        instr2 = mul(N, mask_1, pt2[i6][i2])
        __partial_1[i2][i6] = instr2
    }
}
for i2 in range(3) {
    for i6 in range(3) {
        encode(__partial_1[i2][i6])
    }
}
var __out: C = 0
var __reduce_2: C = 0
for i2 in range(3) {
    var __reduce_1: C = 0
    for i6 in range(3) {
        instr4 = rot(CC, ((-128 * i2) + (-1 * i6)), v_img_1)
        instr6 = mul(CP, instr4, __partial_1[i2][i6])
        instr7 = add(CC, __reduce_1, instr6)
        __reduce_1 = instr7
    }
    instr9 = add(CC, __reduce_2, __reduce_1)
    __reduce_2 = instr9
}
__out = __reduce_2
```

conv-siso e1-o0

```
val v_img_1: C = vector(img(0, 0)[(30, 0, 2 {0 :: 1}), (3, 0, 1
    {0 :: 1}), (32, 0, 0 {1 :: 1})])
val v_filter_1: N = vector(filter(0, 0)[(30, 0, 2, {}), (3, 0,
   1 {0 :: 1}), (30, 0, 2, {})])
```

```
val v_filter_2: N = vector(filter(0, 1)[(30, 0, 2, {}), (3, 0,
   1 {0 :: 1}), (30, 0, 2, {})])
val v_filter_3: N = vector(filter(0, 2)[(30, 0, 2, {}), (3, 0,
   1 {0 :: 1}), (30, 0, 2, {})])
val mask_1: N = mask([(32, 0, 30), (4, 0, 3), (32, 0, 30)])
encode(v_filter_3)
encode(v_filter_2)
encode(v_filter_1)
encode(mask_1)
var pt2: P[3] = 0
pt2[0] = v_filter_1
pt2[1] = v_filter_2
pt2[2] = v_filter_3
var __out: C = 0
var __reduce_1: C = 0
for i4 in range(3) {
    instr1 = rot(CC, (0 + (-1 * i4)), v_img_1)
    instr3 = mul(CP, instr1, mask_1)
    instr5 = mul(CP, instr3, pt2[i4])
    instr6 = add(CC, __reduce_1, instr5)
    __reduce_1 = instr6
}
instr8 = rot(CC, -64, __reduce_1)
instr9 = add(CC, __reduce_1, instr8)
instr10 = rot(CC, -32, instr9)
instr11 = add(CC, instr9, instr10)
__out = instr11
```

conv-siso e2-o1

```
val v_img_1: C = vector(img(0, 0)[(3, 0, 1 {0 :: 1}), (30, 0, 2
    {0 :: 1}), (32, 0, 0 {1 :: 1})])
val v_filter_1: N = vector(filter(0, 0)[(3, 0, 1 {0 :: 1}),
   (30, 0, 2, {}), (30, 0, 2, {})])
val v_filter_2: N = vector(filter(0, 2)[(3, 0, 1 {0 :: 1}),
   (30, 0, 2, {}), (30, 0, 2, {})])
val v_filter_3: N = vector(filter(0, 1)[(3, 0, 1 {0 :: 1}),
   (30, 0, 2, {}), (30, 0, 2, {})])
val mask_1: N = mask([(4, 0, 3), (32, 0, 30), (32, 0, 30)])
val const_neg1: N = const(-1)
var pt2: P[3] = 0
pt2[0] = v_filter_1
pt2[1] = v_filter_3
pt2[2] = v_filter_2
var __partial_1: N[3] = 0
for i0 in range(3) {
    instr2 = mul(N, mask_1, pt2[i0])
    __partial_1[i0] = instr2
}
encode(const_neg1)
for i0 in range(3) {
    encode(__partial_1[i0])
}
var __out: C = 0
var __reduce_1: C = 0
for i0 in range(3) {
```

```
    instr4 = rot(CC, (i0 * -1), v_img_1)
    instr6 = mul(CP, instr4, __partial_1[i0])
    instr7 = add(CC, __reduce_1, instr6)
    __reduce_1 = instr7
}
instr9 = rot(CC, -2048, __reduce_1)
instr10 = add(CC, __reduce_1, instr9)
instr11 = rot(CC, -1024, instr10)
instr12 = add(CC, instr11, instr10)
__out = instr12
```

distance e1-o0

```
val v_point_1: C = vector(point(0)[(64, 0, 0 {0 :: 1})])
val v_tests_1: N = vector(tests.Roll(1,0)(0, 29)[(64, 0, 0 {0
   :: 1})])
val v_tests_2: N = vector(tests.Roll(1,0)(0, 27)[(64, 0, 0 {0
   :: 1})])
val v_tests_3: N = vector(tests.Roll(1,0)(0, 44)[(64, 0, 0 {0
   :: 1})])
val v_tests_4: N = vector(tests.Roll(1,0)(0, 55)[(64, 0, 0 {0
   :: 1})])
val v_tests_5: N = vector(tests.Roll(1,0)(0, 57)[(64, 0, 0 {0
   :: 1})])
val v_tests_6: N = vector(tests.Roll(1,0)(0, 2)[(64, 0, 0 {0 ::
    1})])
val v_tests_7: N = vector(tests.Roll(1,0)(0, 26)[(64, 0, 0 {0
   :: 1})])
val v_tests_8: N = vector(tests.Roll(1,0)(0, 35)[(64, 0, 0 {0
   :: 1})])
val v_tests_9: N = vector(tests.Roll(1,0)(0, 61)[(64, 0, 0 {0
   :: 1})])
val v_tests_10: N = vector(tests.Roll(1,0)(0, 0)[(64, 0, 0 {0
   :: 1})])
val v_tests_11: N = vector(tests.Roll(1,0)(0, 9)[(64, 0, 0 {0
   :: 1})])
val v_tests_12: N = vector(tests.Roll(1,0)(0, 17)[(64, 0, 0 {0
   :: 1})])
val v_tests_13: N = vector(tests.Roll(1,0)(0, 20)[(64, 0, 0 {0
   :: 1})])
val v_tests_14: N = vector(tests.Roll(1,0)(0, 12)[(64, 0, 0 {0
   :: 1})])
val v_tests_15: N = vector(tests.Roll(1,0)(0, 22)[(64, 0, 0 {0
   :: 1})])
val v_tests_16: N = vector(tests.Roll(1,0)(0, 11)[(64, 0, 0 {0
   :: 1})])
val v_tests_17: N = vector(tests.Roll(1,0)(0, 18)[(64, 0, 0 {0
   :: 1})])
val v_tests_18: N = vector(tests.Roll(1,0)(0, 60)[(64, 0, 0 {0
   :: 1})])
val v_tests_19: N = vector(tests.Roll(1,0)(0, 54)[(64, 0, 0 {0
   :: 1})])
val v_tests_20: N = vector(tests.Roll(1,0)(0, 63)[(64, 0, 0 {0
   :: 1})])
val v_tests_21: N = vector(tests.Roll(1,0)(0, 38)[(64, 0, 0 {0
   :: 1})])
```

```
val v_tests_22: N = vector(tests.Roll(1,0)(0, 7)[(64, 0, 0 {0
    :: 1})])
val v_tests_23: N = vector(tests.Roll(1,0)(0, 16)[(64, 0, 0 {0
    :: 1})])
val v_tests_24: N = vector(tests.Roll(1,0)(0, 5)[(64, 0, 0 {0
    :: 1})])
val v_tests_25: N = vector(tests.Roll(1,0)(0, 21)[(64, 0, 0 {0
    :: 1})])
val v_tests_26: N = vector(tests.Roll(1,0)(0, 47)[(64, 0, 0 {0
    :: 1})])
val v_tests_27: N = vector(tests.Roll(1,0)(0, 24)[(64, 0, 0 {0
    :: 1})])
val v_tests_28: N = vector(tests.Roll(1,0)(0, 43)[(64, 0, 0 {0
    :: 1})])
val v_tests_29: N = vector(tests.Roll(1,0)(0, 48)[(64, 0, 0 {0
    :: 1})])
val v_tests_30: N = vector(tests.Roll(1,0)(0, 36)[(64, 0, 0 {0
    :: 1})])
val v_tests_31: N = vector(tests.Roll(1,0)(0, 53)[(64, 0, 0 {0
    :: 1})])
val v_tests_32: N = vector(tests.Roll(1,0)(0, 32)[(64, 0, 0 {0
    :: 1})])
val v_tests_33: N = vector(tests.Roll(1,0)(0, 1)[(64, 0, 0 {0
    :: 1})])
val v_tests_34: N = vector(tests.Roll(1,0)(0, 3)[(64, 0, 0 {0
    :: 1})])
val v_tests_35: N = vector(tests.Roll(1,0)(0, 30)[(64, 0, 0 {0
    :: 1})])
val v_tests_36: N = vector(tests.Roll(1,0)(0, 42)[(64, 0, 0 {0
    :: 1})])
val v_tests_37: N = vector(tests.Roll(1,0)(0, 59)[(64, 0, 0 {0
    :: 1})])
val v_tests_38: N = vector(tests.Roll(1,0)(0, 6)[(64, 0, 0 {0
    :: 1})])
val v_tests_39: N = vector(tests.Roll(1,0)(0, 13)[(64, 0, 0 {0
    :: 1})])
val v_tests_40: N = vector(tests.Roll(1,0)(0, 15)[(64, 0, 0 {0
    :: 1})])
val v_tests_41: N = vector(tests.Roll(1,0)(0, 40)[(64, 0, 0 {0
    :: 1})])
val v_tests_42: N = vector(tests.Roll(1,0)(0, 51)[(64, 0, 0 {0
    :: 1})])
val v_tests_43: N = vector(tests.Roll(1,0)(0, 8)[(64, 0, 0 {0
    :: 1})])
val v_tests_44: N = vector(tests.Roll(1,0)(0, 37)[(64, 0, 0 {0
    :: 1})])
val v_tests_45: N = vector(tests.Roll(1,0)(0, 46)[(64, 0, 0 {0
    :: 1})])
val v_tests_46: N = vector(tests.Roll(1,0)(0, 56)[(64, 0, 0 {0
    :: 1})])
val v_tests_47: N = vector(tests.Roll(1,0)(0, 39)[(64, 0, 0 {0
    :: 1})])
val v_tests_48: N = vector(tests.Roll(1,0)(0, 58)[(64, 0, 0 {0
    :: 1})])
val v_tests_49: N = vector(tests.Roll(1,0)(0, 25)[(64, 0, 0 {0
    :: 1})])
val v_tests_50: N = vector(tests.Roll(1,0)(0, 62)[(64, 0, 0 {0
```

```
    :: 1})])
val v_tests_51: N = vector(tests.Roll(1,0)(0, 28)[(64, 0, 0 {0
    :: 1})])
val v_tests_52: N = vector(tests.Roll(1,0)(0, 34)[(64, 0, 0 {0
    :: 1})])
val v_tests_53: N = vector(tests.Roll(1,0)(0, 23)[(64, 0, 0 {0
    :: 1})])
val v_tests_54: N = vector(tests.Roll(1,0)(0, 10)[(64, 0, 0 {0
    :: 1})])
val v_tests_55: N = vector(tests.Roll(1,0)(0, 14)[(64, 0, 0 {0
    :: 1})])
val v_tests_56: N = vector(tests.Roll(1,0)(0, 33)[(64, 0, 0 {0
    :: 1})])
val v_tests_57: N = vector(tests.Roll(1,0)(0, 4)[(64, 0, 0 {0
    :: 1})])
val v_tests_58: N = vector(tests.Roll(1,0)(0, 41)[(64, 0, 0 {0
    :: 1})])
val v_tests_59: N = vector(tests.Roll(1,0)(0, 50)[(64, 0, 0 {0
    :: 1})])
val v_tests_60: N = vector(tests.Roll(1,0)(0, 52)[(64, 0, 0 {0
    :: 1})])
val v_tests_61: N = vector(tests.Roll(1,0)(0, 49)[(64, 0, 0 {0
    :: 1})])
val v_tests_62: N = vector(tests.Roll(1,0)(0, 19)[(64, 0, 0 {0
    :: 1})])
val v_tests_63: N = vector(tests.Roll(1,0)(0, 45)[(64, 0, 0 {0
    :: 1})])
val v_tests_64: N = vector(tests.Roll(1,0)(0, 31)[(64, 0, 0 {0
    :: 1})])
val const_neg1: N = const(-1)
encode(v_tests_19)
encode(v_tests_48)
encode(v_tests_52)
encode(v_tests_41)
encode(v_tests_54)
encode(v_tests_42)
encode(v_tests_13)
encode(v_tests_4)
encode(v_tests_11)
encode(v_tests_38)
encode(v_tests_15)
encode(v_tests_53)
encode(v_tests_32)
encode(v_tests_29)
encode(v_tests_18)
encode(v_tests_27)
encode(v_tests_58)
encode(v_tests_47)
encode(v_tests_8)
encode(v_tests_63)
encode(v_tests_7)
encode(v_tests_62)
encode(v_tests_16)
encode(v_tests_31)
encode(v_tests_45)
encode(v_tests_50)
encode(v_tests_20)
```

```
encode(v_tests_56)
encode(v_tests_2)
encode(v_tests_43)
encode(v_tests_10)
encode(v_tests_57)
encode(v_tests_6)
encode(v_tests_34)
encode(v_tests_17)
encode(v_tests_61)
encode(v_tests_3)
encode(v_tests_39)
encode(v_tests_26)
encode(v_tests_35)
encode(v_tests_25)
encode(v_tests_46)
encode(v_tests_36)
encode(v_tests_49)
encode(v_tests_22)
encode(v_tests_23)
encode(v_tests_28)
encode(v_tests_30)
encode(v_tests_12)
encode(v_tests_59)
encode(v_tests_40)
encode(v_tests_33)
encode(v_tests_24)
encode(v_tests_1)
encode(v_tests_44)
encode(v_tests_60)
encode(v_tests_21)
encode(v_tests_37)
encode(v_tests_64)
encode(v_tests_51)
encode(v_tests_9)
encode(v_tests_14)
encode(v_tests_55)
encode(v_tests_5)
encode(const_neg1)
var pt1: P[64] = 0
pt1[0] = v_tests_10
pt1[1] = v_tests_33
pt1[2] = v_tests_6
pt1[3] = v_tests_34
pt1[4] = v_tests_57
pt1[5] = v_tests_24
pt1[6] = v_tests_38
pt1[7] = v_tests_22
pt1[8] = v_tests_43
pt1[9] = v_tests_11
pt1[10] = v_tests_54
pt1[11] = v_tests_16
pt1[12] = v_tests_14
pt1[13] = v_tests_39
pt1[14] = v_tests_55
pt1[15] = v_tests_40
pt1[16] = v_tests_23
pt1[17] = v_tests_12
```

```
pt1[18] = v_tests_17
pt1[19] = v_tests_62
pt1[20] = v_tests_13
pt1[21] = v_tests_25
pt1[22] = v_tests_15
pt1[23] = v_tests_53
pt1[24] = v_tests_27
pt1[25] = v_tests_49
pt1[26] = v_tests_7
pt1[27] = v_tests_2
pt1[28] = v_tests_51
pt1[29] = v_tests_1
pt1[30] = v_tests_35
pt1[31] = v_tests_64
pt1[32] = v_tests_32
pt1[33] = v_tests_56
pt1[34] = v_tests_52
pt1[35] = v_tests_8
pt1[36] = v_tests_30
pt1[37] = v_tests_44
pt1[38] = v_tests_21
pt1[39] = v_tests_47
pt1[40] = v_tests_41
pt1[41] = v_tests_58
pt1[42] = v_tests_36
pt1[43] = v_tests_28
pt1[44] = v_tests_3
pt1[45] = v_tests_63
pt1[46] = v_tests_45
pt1[47] = v_tests_26
pt1[48] = v_tests_29
pt1[49] = v_tests_61
pt1[50] = v_tests_59
pt1[51] = v_tests_42
pt1[52] = v_tests_60
pt1[53] = v_tests_31
pt1[54] = v_tests_19
pt1[55] = v_tests_4
pt1[56] = v_tests_46
pt1[57] = v_tests_5
pt1[58] = v_tests_48
pt1[59] = v_tests_37
pt1[60] = v_tests_18
pt1[61] = v_tests_9
pt1[62] = v_tests_50
pt1[63] = v_tests_20
var __out: C = 0
var __reduce_1: C[64] = 0
for i6 in range(64) {
    instr1 = rot(CC, (0 + i6), v_point_1)
    instr3 = sub(CP, instr1, pt1[i6])
    instr4 = mul(CC, instr3, instr3)
    __reduce_1[i6] = instr4
}
instr6 = add(CC, __reduce_1[21], __reduce_1[20])
instr7 = add(CC, __reduce_1[23], __reduce_1[22])
instr8 = add(CC, instr6, instr7)
```

```
instr9 = add(CC, __reduce_1[17], __reduce_1[16])
instr10 = add(CC, __reduce_1[19], __reduce_1[18])
instr11 = add(CC, instr9, instr10)
instr12 = add(CC, instr8, instr11)
instr13 = add(CC, __reduce_1[29], __reduce_1[28])
instr14 = add(CC, __reduce_1[31], __reduce_1[30])
instr15 = add(CC, instr13, instr14)
instr16 = add(CC, __reduce_1[25], __reduce_1[24])
instr17 = add(CC, __reduce_1[27], __reduce_1[26])
instr18 = add(CC, instr16, instr17)
instr19 = add(CC, instr15, instr18)
instr20 = add(CC, instr12, instr19)
instr21 = add(CC, __reduce_1[5], __reduce_1[4])
instr22 = add(CC, __reduce_1[7], __reduce_1[6])
instr23 = add(CC, instr21, instr22)
instr24 = add(CC, __reduce_1[1], __reduce_1[0])
instr25 = add(CC, __reduce_1[3], __reduce_1[2])
instr26 = add(CC, instr24, instr25)
instr27 = add(CC, instr23, instr26)
instr28 = add(CC, __reduce_1[13], __reduce_1[12])
instr29 = add(CC, __reduce_1[15], __reduce_1[14])
instr30 = add(CC, instr28, instr29)
instr31 = add(CC, __reduce_1[9], __reduce_1[8])
instr32 = add(CC, __reduce_1[11], __reduce_1[10])
instr33 = add(CC, instr31, instr32)
instr34 = add(CC, instr30, instr33)
instr35 = add(CC, instr27, instr34)
instr36 = add(CC, instr20, instr35)
instr37 = add(CC, __reduce_1[53], __reduce_1[52])
instr38 = add(CC, __reduce_1[55], __reduce_1[54])
instr39 = add(CC, instr37, instr38)
instr40 = add(CC, __reduce_1[49], __reduce_1[48])
instr41 = add(CC, __reduce_1[51], __reduce_1[50])
instr42 = add(CC, instr40, instr41)
instr43 = add(CC, instr39, instr42)
instr44 = add(CC, __reduce_1[61], __reduce_1[60])
instr45 = add(CC, __reduce_1[63], __reduce_1[62])
instr46 = add(CC, instr44, instr45)
instr47 = add(CC, __reduce_1[57], __reduce_1[56])
instr48 = add(CC, __reduce_1[59], __reduce_1[58])
instr49 = add(CC, instr47, instr48)
instr50 = add(CC, instr46, instr49)
instr51 = add(CC, instr43, instr50)
instr52 = add(CC, __reduce_1[37], __reduce_1[36])
instr53 = add(CC, __reduce_1[39], __reduce_1[38])
instr54 = add(CC, instr52, instr53)
instr55 = add(CC, __reduce_1[33], __reduce_1[32])
instr56 = add(CC, __reduce_1[35], __reduce_1[34])
instr57 = add(CC, instr55, instr56)
instr58 = add(CC, instr54, instr57)
instr59 = add(CC, __reduce_1[45], __reduce_1[44])
instr60 = add(CC, __reduce_1[47], __reduce_1[46])
instr61 = add(CC, instr59, instr60)
instr62 = add(CC, __reduce_1[41], __reduce_1[40])
instr63 = add(CC, __reduce_1[43], __reduce_1[42])
instr64 = add(CC, instr62, instr63)
instr65 = add(CC, instr61, instr64)
```

```
instr66 = add(CC, instr58, instr65)
instr67 = add(CC, instr51, instr66)
instr68 = add(CC, instr36, instr67)
__out = instr68
```

distance e2-o0

```
val v_point_1: C = vector(point(0)[(64, 0, 0, {}), (32, 0, 0 {0
    :: 2})])
val v_point_2: C = vector(point(1)[(64, 0, 0, {}), (32, 0, 0 {0
    :: 2})])
val v_tests_1: N = vector(tests(0, 0)[(64, 0, 0 {0 :: 1}), (32,
    0, 0 {1 :: 2})])
val v_tests_2: N = vector(tests(0, 1)[(64, 0, 0 {0 :: 1}), (32,
    0, 0 {1 :: 2})])
val const_neg1: N = const(-1)
encode(v_tests_2)
encode(v_tests_1)
encode(const_neg1)
var ct1: C[2] = 0
ct1[0] = v_point_1
ct1[1] = v_point_2
var pt1: P[2] = 0
pt1[0] = v_tests_1
pt1[1] = v_tests_2
var __out: C = 0
var __reduce_1: C = 0
for i3i in range(2) {
    instr2 = sub(CP, ct1[i3i], pt1[i3i])
    instr3 = mul(CC, instr2, instr2)
    instr4 = add(CC, __reduce_1, instr3)
    __reduce_1 = instr4
}
instr6 = rot(CC, -16, __reduce_1)
instr7 = add(CC, __reduce_1, instr6)
instr8 = rot(CC, -8, instr7)
instr9 = add(CC, instr7, instr8)
instr10 = rot(CC, -4, instr9)
instr11 = add(CC, instr9, instr10)
instr12 = rot(CC, -2, instr11)
instr13 = add(CC, instr11, instr12)
instr14 = rot(CC, -1, instr13)
instr15 = add(CC, instr13, instr14)
__out = instr15
```

double-matmul e1-o0

```
val v_B_1: C = vector(B(0, 0)[(16, 0, 0 {1 :: 1}), (16, 0, 0 {0
    :: 1}), (16, 0, 0, {})])
val v_A2_1: N = vector(A2(0, 0)[(16, 0, 0, {}), (16, 0, 0 {0 ::
    1}), (16, 0, 0 {1 :: 1})])
val v_A1_1: N = vector(A1(0, 0)[(16, 0, 0, {}), (16, 0, 0 {1 ::
    1}), (16, 0, 0 {0 :: 1})])
val mask_1: N = mask([(16, 0, 15), (16, 0, 0), (16, 0, 15)])
val const_neg1: N = const(-1)
```

```
encode(v_A2_1)
encode(v_A1_1)
encode(mask_1)
encode(const_neg1)
var res: C = 0
instr2 = mul(CP, v_B_1, v_A1_1)
instr3 = rot(CC, -128, instr2)
instr4 = add(CC, instr2, instr3)
instr5 = rot(CC, -64, instr4)
instr6 = add(CC, instr4, instr5)
instr7 = rot(CC, -32, instr6)
instr8 = add(CC, instr6, instr7)
instr9 = rot(CC, -16, instr8)
instr10 = add(CC, instr8, instr9)
res = instr10
var ct2: C[1] = 0
ct2[0] = res
var __circ_1: C[1] = 0
for i in range(1) {
    instr13 = mul(CP, ct2[i], mask_1)
    instr14 = rot(CC, 16, instr13)
    instr15 = add(CC, instr13, instr14)
    instr16 = rot(CC, 32, instr15)
    instr17 = add(CC, instr15, instr16)
    instr18 = rot(CC, 64, instr17)
    instr19 = add(CC, instr17, instr18)
    instr20 = rot(CC, 128, instr19)
    instr21 = add(CC, instr19, instr20)
    __circ_1[i] = instr21
}
var __out: C = 0
instr24 = mul(CP, __circ_1[0], v_A2_1)
instr25 = rot(CC, -8, instr24)
instr26 = add(CC, instr24, instr25)
instr27 = rot(CC, -4, instr26)
instr28 = add(CC, instr26, instr27)
instr29 = rot(CC, -2, instr28)
instr30 = add(CC, instr28, instr29)
instr31 = rot(CC, -1, instr30)
instr32 = add(CC, instr30, instr31)
__out = instr32
```

retrieval-256 e1-o0

```
val v_values_1: C = vector(values(0)[(256, 0, 0 {0 :: 1})])
val v_keys_1: C = vector(keys(0, 0)[(8, 0, 0 {1 :: 1}), (256,
   0, 0 {0 :: 1})])
val v_query_1: C = vector(query(0)[(8, 0, 0 {0 :: 1}), (256, 0,
    0, {})])
val const_1: N = const(1)
val const_neg1: N = const(-1)
encode(const_1)
encode(const_neg1)
var mask: C = 0
instr3 = sub(CC, v_query_1, v_keys_1)
instr4 = mul(CC, instr3, instr3)
```

```
instr5 = mul(CP, instr4, const_neg1)
instr6 = add(CP, instr5, const_1)
instr7 = rot(CC, -1024, instr6)
instr8 = mul(CC, instr6, instr7)
instr9 = rot(CC, -512, instr8)
instr10 = mul(CC, instr8, instr9)
instr11 = rot(CC, -256, instr10)
instr12 = mul(CC, instr10, instr11)
mask = instr12
var __out: C = 0
instr15 = mul(CC, v_values_1, mask)
instr16 = rot(CC, -128, instr15)
instr17 = add(CC, instr15, instr16)
instr18 = rot(CC, -64, instr17)
instr19 = add(CC, instr17, instr18)
instr20 = rot(CC, -32, instr19)
instr21 = add(CC, instr19, instr20)
instr22 = rot(CC, -16, instr21)
instr23 = add(CC, instr21, instr22)
instr24 = rot(CC, -8, instr23)
instr25 = add(CC, instr23, instr24)
instr26 = rot(CC, -4, instr25)
instr27 = add(CC, instr25, instr26)
instr28 = rot(CC, -2, instr27)
instr29 = add(CC, instr27, instr28)
instr30 = rot(CC, -1, instr29)
instr31 = add(CC, instr29, instr30)
__out = instr31
```

retrieval-1024 e1-o0

```
val v_query_1: C = vector(query(6)[(1024, 0, 0, {})])
val v_query_2: C = vector(query(9)[(1024, 0, 0, {})])
val v_query_3: C = vector(query(2)[(1024, 0, 0, {})])
val v_query_4: C = vector(query(5)[(1024, 0, 0, {})])
val v_query_5: C = vector(query(7)[(1024, 0, 0, {})])
val v_keys_1: C = vector(keys(0, 8)[(1024, 0, 0 {0 :: 1})])
val v_query_6: C = vector(query(4)[(1024, 0, 0, {})])
val v_keys_2: C = vector(keys(0, 6)[(1024, 0, 0 {0 :: 1})])
val v_keys_3: C = vector(keys(0, 9)[(1024, 0, 0 {0 :: 1})])
val v_query_7: C = vector(query(8)[(1024, 0, 0, {})])
val v_query_8: C = vector(query(3)[(1024, 0, 0, {})])
val v_keys_4: C = vector(keys(0, 1)[(1024, 0, 0 {0 :: 1})])
val v_keys_5: C = vector(keys(0, 7)[(1024, 0, 0 {0 :: 1})])
val v_query_9: C = vector(query(1)[(1024, 0, 0, {})])
val v_query_10: C = vector(query(0)[(1024, 0, 0, {})])
val v_keys_6: C = vector(keys(0, 5)[(1024, 0, 0 {0 :: 1})])
val v_keys_7: C = vector(keys(0, 0)[(1024, 0, 0 {0 :: 1})])
val v_keys_8: C = vector(keys(0, 2)[(1024, 0, 0 {0 :: 1})])
val v_keys_9: C = vector(keys(0, 3)[(1024, 0, 0 {0 :: 1})])
val v_keys_10: C = vector(keys(0, 4)[(1024, 0, 0 {0 :: 1})])
val v_values_1: C = vector(values(0)[(1024, 0, 0 {0 :: 1})])
val const_1: N = const(1)
val const_neg1: N = const(-1)
encode(const_1)
encode(const_neg1)
```

```
var ct1: C[10] = 0
ct1[0] = v_query_10
ct1[1] = v_query_9
ct1[2] = v_query_3
ct1[3] = v_query_8
ct1[4] = v_query_6
ct1[5] = v_query_4
ct1[6] = v_query_1
ct1[7] = v_query_5
ct1[8] = v_query_7
ct1[9] = v_query_2
var ct2: C[10] = 0
ct2[0] = v_keys_7
ct2[1] = v_keys_4
ct2[2] = v_keys_8
ct2[3] = v_keys_9
ct2[4] = v_keys_10
ct2[5] = v_keys_6
ct2[6] = v_keys_2
ct2[7] = v_keys_5
ct2[8] = v_keys_1
ct2[9] = v_keys_3
var mask: C = 0
var __reduce_1: C[10] = 1
for i5 in range(10) {
    instr3 = sub(CC, ct1[i5], ct2[i5])
    instr4 = mul(CC, instr3, instr3)
    instr5 = mul(CP, instr4, const_neg1)
    instr6 = add(CP, instr5, const_1)
    __reduce_1[i5] = instr6
}
instr8 = mul(CC, __reduce_1[1], __reduce_1[0])
instr9 = mul(CC, __reduce_1[7], __reduce_1[6])
instr10 = mul(CC, __reduce_1[9], __reduce_1[8])
instr11 = mul(CC, instr9, instr10)
instr12 = mul(CC, __reduce_1[3], __reduce_1[2])
instr13 = mul(CC, __reduce_1[5], __reduce_1[4])
instr14 = mul(CC, instr12, instr13)
instr15 = mul(CC, instr11, instr14)
instr16 = mul(CC, instr8, instr15)
mask = instr16
var __out: C = 0
instr19 = mul(CC, v_values_1, mask)
instr20 = rot(CC, -512, instr19)
instr21 = add(CC, instr19, instr20)
instr22 = rot(CC, -256, instr21)
instr23 = add(CC, instr21, instr22)
instr24 = rot(CC, -128, instr23)
instr25 = add(CC, instr23, instr24)
instr26 = rot(CC, -64, instr25)
instr27 = add(CC, instr25, instr26)
instr28 = rot(CC, -32, instr27)
instr29 = add(CC, instr27, instr28)
instr30 = rot(CC, -16, instr29)
instr31 = add(CC, instr29, instr30)
instr32 = rot(CC, -8, instr31)
instr33 = add(CC, instr31, instr32)
```

```
instr34 = rot(CC, -4, instr33)
instr35 = add(CC, instr33, instr34)
instr36 = rot(CC, -2, instr35)
instr37 = add(CC, instr35, instr36)
instr38 = rot(CC, -1, instr37)
instr39 = add(CC, instr37, instr38)
__out = instr39
```

retrieval-1024 e2-o0

```
val v_values_1: C = vector(values(0)[(1024, 0, 0 {0 :: 1})])
val v_query_1: C = vector(query(0)[(8, 0, 0 {0 :: 1}), (1024,
   0, 0, {})])
val v_keys_1: C = vector(keys(0, 8)[(2, 0, 6 {1 :: 1}), (1024,
   0, 0 {0 :: 1})])
val v_query_2: C = vector(query(8)[(2, 0, 6 {0 :: 1}), (1024,
   0, 0, {})])
val v_keys_2: C = vector(keys(0, 0)[(8, 0, 0 {1 :: 1}), (1024,
   0, 0 {0 :: 1})])
val const_1: N = const(1)
val const_neg1: N = const(-1)
encode(const_1)
encode(const_neg1)
var ct2: C[2] = 0
ct2[0] = v_keys_2
ct2[1] = v_keys_1
var ct1: C[2] = 0
ct1[0] = v_query_1
ct1[1] = v_query_2
var mask: C = 0
var __reduce_1: C = 1
for i4o in range(2) {
    instr3 = sub(CC, ct1[i4o], ct2[i4o])
    instr4 = mul(CC, instr3, instr3)
    instr5 = mul(CP, instr4, const_neg1)
    instr6 = add(CP, instr5, const_1)
    instr7 = mul(CC, __reduce_1, instr6)
    __reduce_1 = instr7
}
instr9 = rot(CC, -4096, __reduce_1)
instr10 = mul(CC, __reduce_1, instr9)
instr11 = rot(CC, -2048, instr10)
instr12 = mul(CC, instr10, instr11)
instr13 = rot(CC, -1024, instr12)
instr14 = mul(CC, instr12, instr13)
mask = instr14
var __out: C = 0
instr17 = mul(CC, v_values_1, mask)
instr18 = rot(CC, -512, instr17)
instr19 = add(CC, instr17, instr18)
instr20 = rot(CC, -256, instr19)
instr21 = add(CC, instr19, instr20)
instr22 = rot(CC, -128, instr21)
instr23 = add(CC, instr21, instr22)
instr24 = rot(CC, -64, instr23)
instr25 = add(CC, instr23, instr24)
```

```
instr26 = rot(CC, -32, instr25)
instr27 = add(CC, instr25, instr26)
instr28 = rot(CC, -16, instr27)
instr29 = add(CC, instr27, instr28)
instr30 = rot(CC, -8, instr29)
instr31 = add(CC, instr29, instr30)
instr32 = rot(CC, -4, instr31)
instr33 = add(CC, instr31, instr32)
instr34 = rot(CC, -2, instr33)
instr35 = add(CC, instr33, instr34)
instr36 = rot(CC, -1, instr35)
instr37 = add(CC, instr35, instr36)
__out = instr37
```

set-union-16 e1-o0

```
val v_b_data_1: C = vector(b_data(0)[(16, 0, 0 {0 :: 1})])
val v_a_id_1: C = vector(a_id(0, 0)[(16, 0, 0 {0 :: 1}), (4, 0,
    0 {1 :: 1}), (16, 0, 0, {})])
val v_b_id_1: C = vector(b_id(0, 0)[(16, 0, 0, {}), (4, 0, 0 {1
    :: 1}), (16, 0, 0 {0 :: 1})])
val v_a_data_1: C = vector(a_data(0)[(16, 0, 0 {0 :: 1})])
val const_1: N = const(1)
val const_neg1: N = const(-1)
encode(const_1)
encode(const_neg1)
var b_sum: C = 0
instr4 = sub(CC, v_a_id_1, v_b_id_1)
instr5 = mul(CC, instr4, instr4)
instr6 = mul(CP, instr5, const_neg1)
instr7 = add(CP, instr6, const_1)
instr8 = rot(CC, -32, instr7)
instr9 = mul(CC, instr7, instr8)
instr10 = rot(CC, -16, instr9)
instr11 = mul(CC, instr9, instr10)
instr12 = mul(CP, instr11, const_neg1)
instr13 = add(CP, instr12, const_1)
instr14 = rot(CC, -512, instr13)
instr15 = mul(CC, instr13, instr14)
instr16 = rot(CC, -256, instr15)
instr17 = mul(CC, instr15, instr16)
instr18 = rot(CC, -128, instr17)
instr19 = mul(CC, instr17, instr18)
instr20 = rot(CC, -64, instr19)
instr21 = mul(CC, instr19, instr20)
instr22 = mul(CC, v_b_data_1, instr21)
instr23 = rot(CC, -8, instr22)
instr24 = add(CC, instr22, instr23)
instr25 = rot(CC, -4, instr24)
instr26 = add(CC, instr24, instr25)
instr27 = rot(CC, -2, instr26)
instr28 = add(CC, instr26, instr27)
instr29 = rot(CC, -1, instr28)
instr30 = add(CC, instr28, instr29)
b_sum = instr30
var a_sum: C = 0
```

```
instr32 = rot(CC, -8, v_a_data_1)
instr33 = add(CC, v_a_data_1, instr32)
instr34 = rot(CC, -4, instr33)
instr35 = add(CC, instr33, instr34)
instr36 = rot(CC, -2, instr35)
instr37 = add(CC, instr35, instr36)
instr38 = rot(CC, -1, instr37)
instr39 = add(CC, instr37, instr38)
a_sum = instr39
var __out: C = 0
instr42 = add(CC, a_sum, b_sum)
__out = instr42
```

set-union-128

```
val v_a_id_1: C = vector(a_id(0, 2)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_b_id_1: C = vector(b_id(0, 1)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_b_id_2: C = vector(b_id(0, 6)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_b_id_3: C = vector(b_id(0, 2)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_a_id_2: C = vector(a_id(0, 0)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_a_data_1: C = vector(a_data(0)[(128, 0, 0 {0 :: 1})])
val v_b_data_1: C = vector(b_data(0)[(128, 0, 0 {0 :: 1})])
val v_a_id_3: C = vector(a_id(0, 4)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_a_id_4: C = vector(a_id(0, 6)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_b_id_4: C = vector(b_id(0, 0)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_b_id_5: C = vector(b_id(0, 3)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_b_id_6: C = vector(b_id(0, 4)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_a_id_5: C = vector(a_id(0, 1)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_b_id_7: C = vector(b_id(0, 5)[(128, 0, 0, {}), (128, 0, 0
    {0 :: 1})])
val v_a_id_6: C = vector(a_id(0, 3)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val v_a_id_7: C = vector(a_id(0, 5)[(128, 0, 0 {0 :: 1}), (128,
    0, 0, {})])
val const_1: N = const(1)
val const_neg1: N = const(-1)
encode(const_1)
encode(const_neg1)
var ct2: C[7] = 0
ct2[0] = v_a_id_2
ct2[1] = v_a_id_5
ct2[2] = v_a_id_1
ct2[3] = v_a_id_6
ct2[4] = v_a_id_3
ct2[5] = v_a_id_7
```

```
ct2[6] = v_a_id_4
var ct3: C[7] = 0
ct3[0] = v_b_id_4
ct3[1] = v_b_id_1
ct3[2] = v_b_id_3
ct3[3] = v_b_id_5
ct3[4] = v_b_id_6
ct3[5] = v_b_id_7
ct3[6] = v_b_id_2
var b_sum: C = 0
var __reduce_1: C[7] = 1
for i8 in range(7) {
    instr4 = sub(CC, ct2[i8], ct3[i8])
    instr5 = mul(CC, instr4, instr4)
    instr6 = mul(CP, instr5, const_neg1)
    instr7 = add(CP, instr6, const_1)
    __reduce_1[i8] = instr7
}
instr9 = mul(CC, __reduce_1[1], __reduce_1[0])
instr10 = mul(CC, __reduce_1[3], __reduce_1[2])
instr11 = mul(CC, __reduce_1[5], __reduce_1[4])
instr12 = mul(CC, instr10, instr11)
instr13 = mul(CC, instr9, instr12)
instr14 = mul(CC, __reduce_1[6], instr13)
instr15 = mul(CP, instr14, const_neg1)
instr16 = add(CP, instr15, const_1)
instr17 = rot(CC, -8192, instr16)
instr18 = mul(CC, instr16, instr17)
instr19 = rot(CC, -4096, instr18)
instr20 = mul(CC, instr18, instr19)
instr21 = rot(CC, -2048, instr20)
instr22 = mul(CC, instr20, instr21)
instr23 = rot(CC, -1024, instr22)
instr24 = mul(CC, instr22, instr23)
instr25 = rot(CC, -512, instr24)
instr26 = mul(CC, instr24, instr25)
instr27 = rot(CC, -256, instr26)
instr28 = mul(CC, instr26, instr27)
instr29 = rot(CC, -128, instr28)
instr30 = mul(CC, instr28, instr29)
instr31 = mul(CC, v_b_data_1, instr30)
instr32 = rot(CC, -64, instr31)
instr33 = add(CC, instr31, instr32)
instr34 = rot(CC, -32, instr33)
instr35 = add(CC, instr33, instr34)
instr36 = rot(CC, -16, instr35)
instr37 = add(CC, instr35, instr36)
instr38 = rot(CC, -8, instr37)
instr39 = add(CC, instr37, instr38)
instr40 = rot(CC, -4, instr39)
instr41 = add(CC, instr39, instr40)
instr42 = rot(CC, -2, instr41)
instr43 = add(CC, instr41, instr42)
instr44 = rot(CC, -1, instr43)
instr45 = add(CC, instr43, instr44)
b_sum = instr45
var a_sum: C = 0
```

```
instr47 = rot(CC, -64, v_a_data_1)
instr48 = add(CC, v_a_data_1, instr47)
instr49 = rot(CC, -32, instr48)
instr50 = add(CC, instr48, instr49)
instr51 = rot(CC, -16, instr50)
instr52 = add(CC, instr50, instr51)
instr53 = rot(CC, -8, instr52)
instr54 = add(CC, instr52, instr53)
instr55 = rot(CC, -4, instr54)
instr56 = add(CC, instr54, instr55)
instr57 = rot(CC, -2, instr56)
instr58 = add(CC, instr56, instr57)
instr59 = rot(CC, -1, instr58)
instr60 = add(CC, instr58, instr59)
a_sum = instr60
var __out: C = 0
instr63 = add(CC, a_sum, b_sum)
__out = instr63
```

BIBLIOGRAPHY

[1] https://github.com/scipr-lab/libsnark.

[2] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C Myers, and Elaine Shi. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 740–755, 2021.

[3] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. Helayers: A tile tensors framework for large neural networks on encrypted data. *Proc. Priv. Enhancing Technol.*, 2023(1):325–342, 2023. doi: 10.56553/popets-2023-0020. URL https://doi.org/10.56553/popets-2023-0020.

[4] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[5] Abdelrahaman Aly, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. *SCALE–MAMBA v1.6 : Documentation*, 2019. URL https://homes.esat.kuleuven.be/~nsmart/SCALE.

[6] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–68, 2019.

[7] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*, pages 569–583, July 2015. doi: 10.1109/CSF.2015.42. URL http://www.cs.cornell.edu/andru/papers/flam.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[10] Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. Differentiating a tensor language. *CoRR*, abs/2008.11256, 2020. URL https://arxiv.org/abs/2008.11256.

[11] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 45–56. ACM, 2019. doi: 10.1145/3338469.3358944. URL https://doi.org/10.1145/3338469.3358944.

[12] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In Francesca Palumbo, Michela Becchi, Martin Schulz, and Kento Sato, editors, *Proceedings of the 16th ACM International Conference on Computing Fron-*

tiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019, pages 3–13. ACM, 2019. doi: 10.1145/3310273.3323047. URL https://doi.org/10.1145/3310273.3323047.

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Financial cryptography and data security. chapter Secure Multiparty Computation Goes Live, pages 325–343. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03548-7. doi: 10.1007/978-3-642-03549-4_20. URL http://dx.doi.org/10.1007/978-3-642-03549-4_20.

[14] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[16] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *11th ASIAN Symposium on Programming Languages and Systems, APLAS 2013*, pages 217–232. Springer, 2013. doi: 10.1007/978-3-319-03542-0_16.

[17] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Ma-*

chine Learning Research, pages 812–821. PMLR, 2019. URL http://proceedings.mlr.press/v97/brutzkus19a.html.

[18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. Hycc: Compilation of hybrid protocols for practical secure computation. In *25th ACM Conf. on Computer and Communications Security (CCS)*, page 847–861, New York, NY, USA, 2018. ACM. ISBN 9781450356930. doi: 10.1145/3243734.3243786.

[19] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, pages 143–202, 2000. doi: 10.1007/s001459910006.

[20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001. doi: 10.1109/SFCS.2001.959888.

[21] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*, pages 494–503. ACM, 2002. doi: 10.1145/509907.509980.

[22] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 13–19, 2015.

[23] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 1875–1891. ACM, October 2017. doi: 10.1145/3133956.3134054. URL http://www.cs.cornell.edu/andru/papers/nmifc.

[24] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid mpc protocols. *Cryptology ePrint Archive*, 2023.

[25] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library-seal v2. 1. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 3–18. Springer, 2017.

[26] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

[27] COIN-OR. Cbc - COIN-OR branch and cut. https://www.coin-or.org/Cbc/, accessed 2023-04-11.

[28] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[29] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[30] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Symp. on Security and Privacy*, pages 253–270. IEEE, 2015. doi: 10.1109/SP.2015.23.

[31] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In Stephen N. Freund and Eran Yahav, editors, *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 375–389. ACM, 2021. doi: 10.1145/3453483.3454050.

[32] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1020–1037, 2018.

[33] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In Kathryn S. McKinley and Kathleen Fisher, editors, *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 142–156. ACM, 2019. doi: 10.1145/3314221.3314628.

[34] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In Alastair F. Donaldson and Emina Torlak, editors, *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 546–561. ACM, 2020. doi: 10.1145/3385412.3386023.

[35] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992. doi: 10.1007/978-3-540-78800-3_24.

[36] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. In *19th European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014. doi: 10.1007/978-3-319-11212-1_18.

[37] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security Symp.* The Internet Society, 2015. doi: 10.14722/ndss.2015.23113. URL https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation.

[38] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976. URL https://dl.acm.org/citation.cfm?id=360056.

[39] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 73–84, 2013. URL http://doi.acm.org/10.1145/2508859.2516693.

[40] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[41] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, PLDI '93, pages 237–247, 1993. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL http://dx.doi.org/10.1145/155090.155113.

[42] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *35th ACM Symp. on Principles of Programming*

Languages (POPL), pages 323–335, January 2008. URL https://doi.org/10.1145/1328438.1328478.

[43] Cédric Fournet, Guervan le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *16th ACM Conf. on Computer and Communications Security (CCS)*, pages 432–441, November 2009. URL https://doi.org/10.1145/1653662.1653715.

[44] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ansi c compiler for secure two-party computations æ. In *Compiler Construction: 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8409, page 244. Springer, 2014.

[45] GDPR. General data protection regulation, 2016. URL https://gdpr-info.eu.

[46] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 38–49. ACM, 2012. doi: 10.1145/2382196.2382204. URL https://doi.org/10.1145/2382196.2382204.

[47] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[48] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982. doi: 10.1109/SP.1982.10014. URL https://ieeexplore.ieee.org/document/6234468.

[49] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In Alfred V. Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, 1987. doi: 10.1145/28395.28420. URL https://doi.org/10.1145/28395.28420.

[50] Anitha Gollamudi and Stephen Chong. Automatic enforcement of expressive security policies using enclaves. In *International Conference on Object-Oriented Programming, Systems, Language & Applications (OOPSLA)*, 2016.

[51] Anitha Gollamudi, Stephen Chong, and Owen Arden. Information flow control for distributed trusted execution environments. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, pages 304–318. IEEE, 2019. doi: 10.1109/CSF.2019.00028. URL https://doi.org/10.1109/CSF.2019.00028.

[52] Mark Gurman. Samsung bans chatgpt, google bard, other generative ai use by staff after leak, May 2023. URL https://www.bloomberg.com/news/articles/2023-05-02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak.

[53] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014. doi: 10.1007/978-3-662-44371-2_31. URL https://doi.org/10.1007/978-3-662-44371-2_31.

[54] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[55] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *IEEE Symp. on Security and Privacy*, pages 1220–1237, 2019. doi: 10.1109/SP.2019.00028. URL https://doi.org/10.1109/SP.2019.00028.

[56] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062354. URL https://doi.org/10.1145/3140587.3062354.

[57] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *26th ACM Conf. on Computer and Communications Security (CCS)*, pages 1539–1556. ACM, 2019. doi: 10.1145/3319535.3339818.

[58] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1651–

1669. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[59] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.

[60] Florian Kerschbaum. Automatically optimizing secure computation. In *18th ACM Conf. on Computer and Communications Security (CCS)*, October 2011. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046786.

[61] G. Kildall. A unified approach to global program optimization. In *ACM Symp. on Principles of Programming Languages (POPL)*, 1973.

[62] Donghwan Kim, Jaiyoung Park, Jongmin Kim, Sangpyo Kim, and Jung Ho Ahn. Hyphen: A hybrid packing method and optimizations for homomorphic encryption-based neural networks. *CoRR*, abs/2302.02407, 2023. doi: 10.48550/arXiv.2302.02407. URL https://doi.org/10.48550/arXiv.2302.02407.

[63] Miran Kim, Arif Ozgun Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, et al. Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation. *Cell systems*, 12(11):1108–1120, 2021.

[64] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *IEEE Symp. on Security and Privacy*, pages 944–961. IEEE, 2018. doi: 10.1109/SP.2018.00018.

[65] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *Acm Sigplan Notices*, 35(5):145–156, 2000.

[66] Kristin Lauter, Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. Password monitor: Safeguarding passwords in microsoft

edge, 2021. URL https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge.

[67] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 503–518, 2020.

[68] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. HECATE: performance-aware scale optimization for homomorphic encryption compiler. In Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, pages 193–204. IEEE, 2022. doi: 10.1109/CGO53902.2022.9741265. URL https://doi.org/10.1109/CGO53902.2022.9741265.

[69] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, P Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel sgx. USENIX, 2017.

[70] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi: 10.1145/3498717. URL https://doi.org/10.1145/3498717.

[71] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *IEEE Symp. on Security and Privacy*, pages 623–638. IEEE, 2014. doi: 10.1109/SP.2014.46.

[72] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm:

A programming framework for secure computation. In *25th ACM Symp. on Operating System Principles (SOSP)*, pages 359–376. IEEE, 2015. doi: 10.1109/SP.2015.29. URL https://doi.org/10.1109/SP.2015.29.

[73] Raghav Malik, Vidush Singhal, Benjamin Gottfried, and Milind Kulkarni. Vectorized secure evaluation of decision forests. In Stephen N. Freund and Eran Yahav, editors, *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 1049–1063. ACM, 2021. doi: 10.1145/3453483.3454094. URL https://doi.org/10.1145/3453483.3454094.

[74] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 118–133, 2023.

[75] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *13th Usenix Security Symposium*, pages 287–302, August 2004. URL http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html.

[76] Charith Mendis and Saman Amarasinghe. goslp: globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[77] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022.

[78] Daniel Morales, Isaac Agudo, and Javier Lopez. Private set intersection: A systematic literature review. *Computer Science Review*, 49:100567, 2023.

[79] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999. doi: 10.1145/292540.292561. URL http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf.

[80] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symp. on Security and Privacy*, pages 186–197, May 1998. URL http://www.cs.cornell.edu/andru/papers/sp98/sp98.pdf.

[81] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

[82] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.

[83] Alex Ozdemir, Fraser Brown, and Riad S Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266. IEEE, 2022.

[84] Shweta Panditrao and Mustafa Emre Acer. A safer default for navigation: Https, Mar 2021. URL https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html.

[85] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symp. on Security and Privacy*, pages 238–252. IEEE, 2013. doi: 10.1109/SP.2013.47.

[86] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: index sets and parallelism-preserving autodiff for point-

ful array programming. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi: 10.1145/3473593. URL https://doi.org/10.1145/3473593.

[87] François Pottier and Vincent Simonet. Information flow inference for ML. In *29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002. doi: 10.1145/503272.503302.

[88] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

[89] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symp. on Security and Privacy*, pages 655–670, May 2014. doi: 10.1109/SP.2014.48.

[90] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.

[91] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996. doi: 10.1007/3-540-61739-6_48.

[92] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture

for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1295–1309, 2020.

[93] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[94] Daniel Edwin Rutherford. *Introduction to Lattice Theory*. Oliver and Boyd, 1965.

[95] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. doi: 10.1109/JSAC.2002.806121. URL http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf.

[96] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic conference on Information Security Technology for Applications*, pages 223–239, 2011. URL www.scs.stanford.edu/~dm/home/papers/stefan:dclabels.pdf.

[97] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang, editors, *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 74–85. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3049841.

[98] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In Zhong Shao and Benjamin C.

Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 264–276. ACM, 2009. doi: 10.1145/1480881.1480915. URL https://doi.org/10.1145/1480881.1480915.

[99] Alexander J Titus, Shashwat Kishore, Todd Stavish, Stephanie M Rogers, and Karl Ni. Pyseal: A python wrapper implementation of the seal homomorphic encryption library. *arXiv preprint arXiv:1803.01891*, 2018.

[100] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX Annual Technical Conference*, pages 645–658, 2017.

[101] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. Sealion: a framework for neural network inference on encrypted data. *CoRR*, abs/1904.12840, 2019. URL http://arxiv.org/abs/1904.12840.

[102] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. URL http://arxiv.org/abs/1802.04730.

[103] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1092–1108. IEEE, 2021. doi: 10.1109/SP40001.2021.00068. URL https://doi.org/10.1109/SP40001.2021.00068.

[104] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: au-

tomatic code optimizations for efficient fully homomorphic encryption. *CoRR*, abs/2202.01649, 2022. URL https://arxiv.org/abs/2202.01649.

[105] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 3:1–3:18, 2019. doi: 10.1145/3302424.3303982. URL https://doi.org/10.1145/3302424.3303982.

[106] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symp.* The Internet Society, 2015. doi: 10.14722/ndss.2015.23097. URL https://www.ndss-symposium.org/ndss2015/efficient-ram-and-control-flow-verifiable-outsourced-computation.

[107] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145/3434304.

[108] Andrew C. Yao. Protocols for secure computations. In *23rd annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982. doi: 10.1109/SFCS.1982.38. URL https://doi.org/10.1109/SFCS.1982.38.

[109] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019. doi: 10.1109/CSF.2019.00026. URL https://www.cs.cornell.edu/andru/papers/hyperisa.

[110] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious

computation. *IACR Cryptol. ePrint Arch.*, 2015. URL http://eprint.iacr.org/2015/1153.

[111] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, June 2001. doi: 10.1109/CSFW.2001.930133. URL http://www.cs.cornell.edu/andru/papers/csfw01.pdf.

[112] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, August 2002. doi: 10.1145/566340.566343. URL http://www.cs.cornell.edu/andru/papers/sosp01/spp-tr.pdf.

[113] Lantian Zheng and Andrew C. Myers. A language-based approach to secure quorum replication. In *9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, August 2014. doi: 10.1145/2637113.2637117. URL http://www.cs.cornell.edu/andru/papers/plas14.

[114] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symp. on Security and Privacy*, pages 236–250, May 2003. doi: 10.1109/SECPRI.2003.1199340. URL http://www.cs.cornell.edu/andru/papers/sp03.pdf.