Abstract

Practical and Deployable Secure Multi-Party Computation

Debayan Gupta

2016

The advent of pervasive computation and the internet has resulted in a world in which a vast amount of private information resides in computers and networks. Unfortunately, recent events have proven that this information is unsafe. In just the last two years, retailers such as Target and Ebay, government agencies such as the Internal Revenue Service and the Office of Personnel Management, and companies as diverse as Ashley Madison (a Canadian dating company specializing in extramarital relationships) and JP Morgan Chase (the largest bank in the US) have all been hacked. Add in the revelations about large-scale spying by government agencies and state-sponsored hacking, and our security is very much in doubt.

Secure function evaluation (SFE), also known as secure multi-party computation (SMPC), allows multiple parties to jointly compute a function while maintaining input and output privacy. The two-party variant, known as 2P-SFE, was first introduced by Yao in the 1980s [1] and was largely a theoretical curiosity. 2P-SFE has become vastly more efficient in recent years, owing to advances in hardware, faster encryption libraries, and, of course, improved protocols [2, 3, 4]. Given all of the threats to our private information, strong cryptography and secure computation should be ubiquitous. However, various technical, economic, and social factors have resulted in a situation in which cryptographic and security technology is not nearly as widely used as it should be.

In this thesis, I present a set of results aimed at making SFE more suitable for real-life deployment. First, I present PartialGC, which fundamentally changes the

communication pattern of SFE from a complex many-to-many pattern to a simple, star-like pattern and introduces the ability to save state across multiple secure computations. To facilitate the creation of secure programs, I then present a complete garbled-circuit compiler for 2P-SFE computations, called Frigate, that is two orders of magnitude faster than the previous state of the art. Next, I combine these technologies with Intel's new Software Guard eXtensions to achieve security guarantees akin to that of SFE while achieving execution speeds almost as fast as those of privacy-invasive protocols. Finally, I present a tool that allows naïve users to search and specify their security needs and assumptions and produces a list of known cryptographic protocols suited for the scenario specified or indicates that no such protocols are known.

Practical and Deployable Secure Multi-Party Computation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Debayan Gupta

Dissertation Director: Joan Feigenbaum

December, 2016

Contents

List of Figures

List of Tables

To Upendramohan Sen

Acknowledgements

A doctoral thesis is the culmination of many years of study and effort, and while the pages that follow summarize the results of that effort, they do little to recognize the many people who stood by my side and made all of this possible.

First, I want to thank my advisor, the extraordinary Joan Feigenbaum. It has been an absolute pleasure working with and learning from her. No words of gratitude can possibly equal my parents' unceasing support, but I say this regardless: thank you. Thanks to all my co-authors, with whom I look forward to working on many, many more projects. Last, but not least, I want to thank my friends, Subhajyoti Chaudhuri, Akshay Deshmukh, Corina Grigore, James Gutierrez, Fabian Schrey, Aaron Segal, Andrew Sinclair, Carol Suh, Andrey Tolstoy, Jeremy Willsey, and Lauren Young. Veræ amicitiæ sempiternæ sunt.

The research in this dissertation was supported by a Computer Science Department Kempner Fellowship and by grants from the Intelligence Advanced Research Projects Activity and the Defense Advanced Research Projects Agency.

Glossary

2P-SFE Two-party secure function evaluation

HBC Honest but curious

SFE Secure function evaluation

SMPC Secure multi-party computation

Chapter 1

Introduction

Computers and networks now mediate a tremendous amount of our daily activity, ranging from shopping to health care to political engagement and even to friendship and romance. This trend is expected to continue and even to accelerate for the foreseeable future. The primary currency of the internet – private information used by advertisers and others to gain insight into our lives – is a dangerous one and has meant that large companies devote significant resources to the procurement and analysis of such data. The economic benefits of this system, however, have ensured that we will continue to create and transmit huge amounts of sensitive data about individuals and organizations.

Secure Multi-Party Computation (SMPC), also known as Secure Function Evaluation (SFE), has long been regarded as a theoretical curiosity. First proposed by Yao [1], SMPC allows two or more parties to compute the result of a function without exposing their inputs. The identification of such primitives was groundbreaking, creating opportunities by which untrusting participants could calculate results of mutual interest without requiring all individuals to identify a mutually trusted third party. Unfortunately, it would take more than 20 years before the creation of the first SMPC compiler, Fairplay [5], demonstrated that these heavyweight techniques were remotely

practical.

The creation of the Fairplay compiler ignited the research community. In the following decade, SMPC compilers improved performance by multiple orders of magnitude, significantly reduced bandwidth overhead, and allowed for the generation and execution of circuits composed of tens of billions of gates [3,6,7,8]. These efforts have brought SMPC from the realm of mere theoretical interest to the verge of practicality; as an indicator of this fundamental change, DARPA is spending $60 million to support the transition of technologies such as SMPC to practice [9].

In principle, we could obviate the need to transfer our sensitive information to organizations that may then be hacked by using SMPC protocols. SMPC enables data owners to *use data without revealing them*. Specifically, it allows parties $P_1, \ldots, P_n$ with private inputs $x_1, \ldots, x_n$ to compute a value $y = f(x_1, \ldots, x_n)$ in such a way that every P_i learns the output y, but no P_i learns anything about x_j, except what is implied by y and x_i. Thus, customers could purchase items without revealing their credit-card numbers or get insurance quotes without revealing their health records by engaging with these companies (and perhaps with other parties, *e.g.*, credit-card issuers) in SMPC protocols. The customers' private information could remain on their personal devices, entirely under their control.

The case for widespread deployment of SMPC has never been stronger. Moreover, recent advances have made SMPC protocols considerably more efficient. In particular, there have been striking improvements in the efficiency of secure, two-party protocols, also known as 2P-SFE ("two-party secure function evaluation"). In this thesis, we address the question of speeding up and improving SMPC to make it practical for everyday usage.

Many fast and powerful SMPC platforms exist today [7, 8, 10, 11, 12, 13, 14], and their performance is improving. Such platforms have been used for scenarios as varied as those of farmers conducting beet-root auctions [15], inter-domain routing [16], gov-

ernments reporting aggregated salary data [17], and database policy compliance [18].

While it is still true that performing a computation using SMPC is usually much slower than doing so in a non-privacy-preserving manner, modern SMPC protocols, especially 2P-SFE protocols, are more than fast enough to solve some real-world problems, even in scenarios with malicious adversaries. Ideally, SMPC could be used in any scenario where the parties trust each other enough to want to cooperate in the first place but not enough to release private data or trust the other parties not to cheat [19].

Unfortunately, people are reluctant to trust cryptographic mechanisms, preferring instead to rely on trusted third parties with existing accountability mechanisms [19]. They are far more likely to go with a tried and trusted system rather than a new (to them) cryptographic protocol, however secure and efficient. In this thesis, we present a set of tools and protocols intended to make SMPC feasible for real-world use. To achieve this we describe four lines of research that together create a complete ecosystem capable of supporting everyday users:

1. First, we introduce "PartialGC," which fundamentally changes the communication pattern of SMPC protocols from a complex many-to-many pattern to the simple, star-like, client-server pattern that is currently used for the vast majority of internet applications. It also introduces the ability to securely save state across multiple secure computations. This makes it feasible for programmers to convert privacy-invasive applications to privacy-preserving ones without having to completely change their product. It also significantly increases the efficiency of many protocols. To illustrate the speed and power of this system, we build and demonstrate the first smartphone application capable of running SMPC protocols. This material appeared in preliminary form in [20].

2. Next, we present "Frigate," an SMPC compiler that allows programmers to write code in a C-like language and produces fast, secure applications with-

out requiring that the programmer be knowledgeable in the underlying cryptographic protocols. This means that programmers will not need to undergo special training in order to write privacy-preserving applications. Current SMPC compilers are mostly intended for research use and are often incomplete, incorrect, or unstable, leading to demonstrably erroneous results and inefficiencies. While industry-standard compilers also contain errors, the number and magnitude of problems we found in most popular SMPC compilers makes them very difficult to use. Frigate has been tested extensively for correctness using industry-standard compiler testing techniques and is two orders of magnitude faster than other state of the art SMPC compilers. This material appeared in preliminary form in [21].

3. We then combine these technologies with Intel's Software Guard Extensions, "SGX," to achieve speeds of execution comparable to that of applications built without privacy in mind, virtually removing the performance overheads involved in converting an insecure application to one that preserves privacy. This material appeared in preliminary form in [22].

4. Finally, most users and businesses would find it difficult, if not impossible, to decide which SMPC protocols to use for their needs, given the huge and growing size of the field [23]. We present a tool, SysSC-UI, which allows naïve users to search and specify their security needs and assumptions. The tool produces a list of known cryptographic protocols suited for the scenario specified or indicates that no such protocols are known. It has also proved to be very useful to researchers trying to find gaps in the literature, which could indicate the existence of an unexplored scenario or an as-yet unproven impossibility result for that scenario. The tool makes use of a systematization of the current state of SMPC and a corresponding annotated bibliography. This material

appeared in preliminary form in [23].

Together, these innovations will allow individuals and businesses to use powerful cryptographic technology and build privacy-preserving applications without making huge investments into security. I believe the eventual aim of all research should be to better understand and improve the world in which we live. Building technologies to safeguard the privacy and security of individuals and institutions is essential to a safe existence in today's connected world full of sensitive information.

Chapter 2

Background

Secure function evaluation (SFE) addresses scenarios where two or more mutually distrustful parties $P_1, \ldots, P_n$, with private inputs $x_1, \ldots, x_n$, want to compute a given function $y_i = f(x_1, \ldots, x_n)$ (y_i is the output received by P_i), such that no P_i learns anything about any x_j or y_j, $i \neq j$ that is not logically implied by x_i and y_i. Moreover, there exists no trusted third party – if there was, the P_is could simply send their inputs to the trusted party, which would evaluate the function and return the y_is.

SFE was first proposed in the 1980s in Yao's seminal paper [1]. The area has been studied extensively by the cryptography community, leading to the creation of the first general purpose platform for SFE, Fairplay [5] in the early 2000s. Today, there exist many such platforms [7, 8, 10, 11, 12, 13, 14]. While these protocols are still much slower than their privacy-invasive counterparts, many programs, especially those operating in the two-party model (called 2P-SFE), are now fast enough to be deployed in real-world scenarios.

2.1 Garbled Circuits

The classic platforms for 2P-SFE, including Fairplay, use garbled circuits. A garbled circuit is a Boolean circuit which is encrypted in such a way that it can be evaluated

when the proper input wires are entered. The party that evaluates this circuit does not learn anything about what any particular wire represents. In 2P-SFE, the two parties are: the *generator*, which creates the garbled circuit, and the *evaluator*, which evaluates the garbled circuit. Additional cryptographic techniques are used for input and output; we discuss these later.

A two-input Boolean gate has four truth table entries. A two-input garbled gate also has a truth table with four entries representing 1s and 0s, but these entries are encrypted and can only be retrieved when the proper keys are used. The values that represent the 1s and 0s are random strings of bits. The truth table entries are permuted such that the evaluator cannot determine which entry she is able to decrypt, only that she is able to decrypt an entry. The entirety of a garbled gate is the four encrypted output values.

For each wire i in the garbled circuit, the generator selects random encryption keys k_i^0, k_i^1 to represent the bit values "0" and "1" for each wire in the circuit. Given these garbled wire labels, each gate in the circuit is represented as a truth table (while each gate may have an arbitrary number of input wires, we assume each gate has two inputs without loss of generality). For a gate executing the functionality $\star$ with input wires i and j and output wire k, the generator encrypts each entry in the truth table as $Enc((k_i^{b_i}, k_j^{b_j}), k_k^{b_i \star b_j})$ where b_i and b_j are the logical bit values of wires i and j. After permuting the entries in each truth table, the generator sends the garbled circuit, along with the input wire labels corresponding to his input, to the evaluator. Given this garbled representation, the evaluator can iteratively decrypt the output wire label for each gate. Once the evaluator possesses wire labels for each output wire, the generator can reveal the actual bit value mapped to the output wire labels received.

To initiate evaluation, the evaluator must hold garbled representations of both parties' input values. However, since the evaluator does not know the mapping be-

tween real bit values and garbled wire labels, we require something that allows the evaluator to garble her own input without revealing it to the generator. To achieve this, we use 1-out-of-2 oblivious transfers (OTs) [24, 25, 26, 27]. In a 1-out-of-2 OT, one party offers up two possible values while the other party selects one of the two values without learning the other. The party that offers up the two values does not learn which value was selected. Using this technique, the evaluator gets the wire labels for her input without leaking information. The only way for the evaluator to get a correct output value from a garbled gate is to know the correct decryption keys for a specific entry in the truth table, as well as the location of the value she has to decrypt.

During the permutation stage, rather than simply randomly permuting the values, the generator permutes values based on a specific bit in $input_x$ and $input_y$, such that, given $input_x$ and $input_y$ the evaluator knows that the location of the entry to decrypt is $bit_x * 2 + bit_y$. These bits are called the *permutation bits*, as they show the evaluator which entry to select based on the permutation; this optimization, which does not leak any information, is known as *point and permute* [5].

This protocol guarantees privacy of both parties' inputs and correctness of the output in the semi-honest adversary model, which assumes that both parties will follow the protocol as specified, and will only try to learn additional information through passive observation. When adversaries can perform arbitrary malicious actions, a number of additional checks must be added to ensure that neither party can break the security of the protocol. These checks are designed specifically to prevent tampering with the evaluated function, providing incorrect or inconsistent inputs, or corrupting the values output by the garbled circuit protocol.

2.2 Threat Models and Definitions

Traditionally, there are two threat models discussed in SFE work, semi-honest and malicious. The above description of garbled circuits is the same in both threat models. In the semi-honest model users stay true to the protocol but may attempt to learn extra information from the system by looking at any message that is sent or received. In the malicious model, users may attempt to change anything with the goal of learning extra information or giving incorrect results without being detected; extra techniques must be added to achieve security against a malicious adversary.

Definition 1. *Given n parties $P_1, \ldots, P_n$ with secret inputs $x_1, \ldots, x_n$, we wish to compute a public function f to get outputs $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$.*

We consider an SFE protocol to be secure and correct if each P_i learns y_i but nothing about $y_j, x_j \forall j \neq i$ beyond what is implied about any x_j by the output y_i. A malicious adversary may corrupt all but one party.

For garbled circuits to achieve security against malicious adversaries, the computation must be performed N times in order to prevent the generator from creating an incorrect circuit. The security parameter N sets the upper bound on an adversary's successfully cheating at $\frac{1}{2^N}$. Crucially, because the computation is performed multiple times, there must be mechanisms to ensure that the same inputs are used each time and a way to ensure the evaluator does not corrupt the generator's output. These are solved problems in the garbled-circuit literature.

There are several well-known attacks a malicious adversary could use against a garbled circuit protocol. A protocol secure against malicious adversaries must have solutions to all potential pitfalls. We briefly describe some of the most important issues below:

Generation of incorrect circuits: If the generator does not create a correct garbled circuit, he could learn extra information by modifying truth table values to output the

evaluator's input; he is limited only by the external structure of the garbled circuit the evaluator expects.

Selective failure of input: If the generator does not offer up correct input wires to the evaluator, and the evaluator selects the wire that was not created properly, the generator can learn up to a single bit of information based on whether the computation produced correct outputs.

Input consistency: If either party's input is not consistent across all circuits, then it might be possible for extra information to be retrieved.

Output consistency: In the two-party case, the output consistency check verifies that the evaluator did not modify the generator's output before sending it.

Chapter 3

Related Work

SMPC was first described by Yao in his seminal paper [1] on the subject. The first general purpose platform for SMPC, Fairplay [5], was created in 2004. Fairplay had both a compiler and a runtime system and revitalized research in the area. Computations involving three or more parties have also been examined; one of the earliest examples is FairplayMP [28]. There have been multiple other implementations since, in both semi-honest [7, 11, 12, 13, 14] and malicious settings [8, 10]. Canetti *et al.* [29] introduced universally composable security, which guarantees security regardless of the execution environment or composition with other protocol executions.

In particular, the garbled circuit protocol has been vastly expanded from its original capability and applied to various areas [16,18], allowing for security in the presence of covert [30], malicious [4, 10, 31, 32, 33], and other adversaries [34], as well as outsourced execution for computationally limited devices [35,36,37,38,39]. Optimizations for garbled circuits include the free-XOR technique [40], garbled row reduction [41], rewriting computations to minimize SMPC [42], and pipelining [2]. Pipelining allows the evaluator to proceed with the computation while the generator is creating gates. Kreuter *et al.* [3] included both an optimizing compiler and an efficient runtime system using a parallelized implementation of SMPC in the malicious model from [10].

We use many of these optimizations in later chapters.

There exist many other methods for performing SMPC, such as homomorphic encryption [43, 44], secret sharing [45], and ordered binary decision diagrams [46]. A general privacy-preserving computation protocol that uses homomorphic encryption and was designed specifically for mobile devices can be found in [39]. Kamara et al. [47] showed how to scale server-aided Private Set Intersection to billion-element sets with a custom protocol.

Despite such advances, SMPC has not seen widespread real-world usage. While Fairplay and other compilers ignited the research community, the only instances of SMPC in real life remain limited to sugar-beet auctions in Denmark [48] and the production of salary statistics for Estonian government employees [49]. In the following chapters, we provide a number of tools that, taken together, will hopefully ease the transition of SMPC from universities and research institutes into the real-world.

Chapter 4

Partial Garbled Circuits

4.1 Introduction

This chapter presents PartialGC, an SFE system that facilitates the reuse of encrypted values generated during a garbled-circuit computation. This allows us to fundamentally change the communication pattern from the many-to-many communications required for most privacy-preserving protocols to a traditional client-server communication pattern common to most everyday applications.

While 2P-SFE has become significantly more feasible, even on resource-constrained devices, there are still bottlenecks, particularly in the input validation stage of a computation. This means that the move from many-to-many to a client-server model also significantly reduces this input-validation overhead. Further, saving state across computations means that expensive processing does not have to be repeated if a similar computation is done again. The reuse of previous inputs to allow stateful evaluation represents a new way of looking at SFE and further reduces computational barriers.

We show that using PartialGC can reduce computation time by as much as 96% and bandwidth by as much as 98% in comparison with previous outsourcing schemes for secure computation. We demonstrate the feasibility of our approach with two

sets of experiments, one in which the garbled circuit is evaluated on a mobile device and one in which it is evaluated on a server. We also use PartialGC to build a privacy-preserving "friend finder" application for Android. This is the first non-trivial SFE-application capable of running on a smartphone.

As mobile devices become more powerful and ubiquitous, users expect more services to be accessible through them. When SFE is performed on mobile devices (where resource constraints are tight), it is extremely slow – *if* the computation can be run at all without exhausting the memory, which can happen for non-trivial input sizes and algorithms [36]. One way to allow mobile devices to perform SFE is to use a server-aided computational model [35,36], allowing the majority of an SFE computation to be "outsourced" to a more powerful device while still preserving privacy. Past approaches, however, have not considered the ways in which mobile computation differs from the desktop. Often, the mobile device is called upon to perform *incremental* operations that are continuations of a previous computation.

Consider, for example, a *friend finder* application where the location of users is updated periodically to determine whether a contact is in proximity. Traditional applications disclose location information to a central server. A privacy-preserving *friend finder* could perform these operations in an oblivious fashion. However, every incremental location update would require a full re-evaluation of the function with fresh inputs in a standard SFE solution. Our examination of an outsourced SFE scheme for mobile devices by Carter et al. [36] (hereon CMTB), determined that the cryptographic consistency checks performed on the inputs to an SFE computation can themselves be the greatest bottleneck to performance.

Additionally, many other applications require the ability to save state, a feature that current garbled circuit implementations do not possess. The ability to save state and reuse an intermediate value from one garbled circuit execution to another would be useful in many other ways, *e.g.*, we could split a large computation into a number

of smaller pieces. Combined with efficient input validation, this becomes an extremely attractive proposition.

In this chapter, we show that it is possible to reuse an encrypted value in an outsourced SFE computation (we use a cut-and-choose garbled circuit protocol) even if one is restricted to primitives that are part of standard garbled circuits. Our system, PartialGC, which is based on CMTB, provides a way to take encrypted output wire values from one SFE computation, save them, and then reuse them as input wires in a new garbled circuit. Our method vastly reduces the number of cryptographic operations compared to the trivial mechanism of simply XOR'ing the results with a one-time pad, which requires either generating inside the circuit, or inputting, a very large one-time pad, both complex operations. Through the use of improved input validation mechanisms proposed by shelat and Shen [4] (hereon sS13) and new methods of *partial input* gate checks and evaluation, we improve on previous proposals. There are other approaches to the creation of reusable garbled circuits [50,51,52], and previous work on reusing encrypted values in the ORAM model [53,54,55], but these earlier schemes have not been implemented. By contrast, we have implemented our scheme and found it to be both practical and efficient; we provide a performance analysis and a sample application to illustrate its feasibility (Section 4.5), as well as a simplified example execution (Appendix A.3).

By breaking a large program into smaller pieces, our system allows interactive I/O throughout the garbled circuit computation. To the best of our knowledge this is the first practical protocol for performing interactive I/O in the middle of a cut-and-choose garbled circuit computation.

Our system comprises three parties - a generator, an evaluator, and a third party ("the cloud"), to which the evaluator outsources its part of the computation. Our protocol is secure against a malicious adversary, assuming that there is no collusion by either party with the cloud. We also provide a semi-honest version of the protocol.

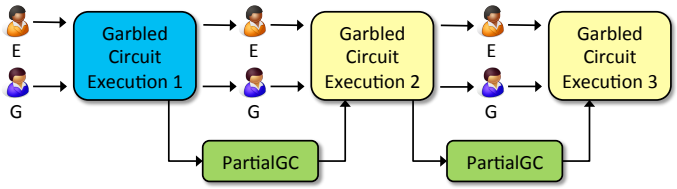


Figure 4.1: PartialGC Overview. E is evaluator and G is generator. The blue box is a standard execution that produces partial outputs (garbled values); yellow boxes represent executions that take partial inputs and produce partial outputs.

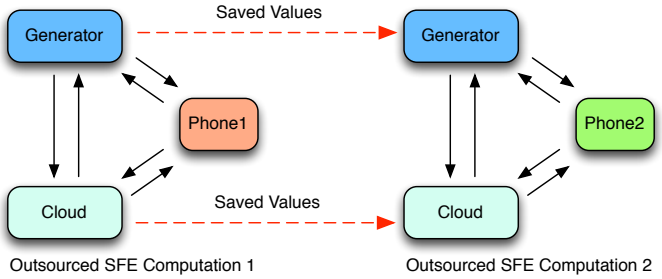


Figure 4.2: Our system has three parties. Only the cloud and generator have to save intermediate values - this means that we can have different phones in different computations.

Figure 4.1 shows how PartialGC works at a high level: First, a standard SFE execution (blue) takes place, at the end of which we "save" some intermediate output values. All further executions use intermediate values from previous executions. In order to reuse these values, information from both parties – the generator and the evaluator – has to be saved. In our protocol, it is the cloud – rather than the evaluator – that saves information. This allows multiple distinct evaluators to participate in a large computation over time by saving state in the cloud between different garbled circuit executions. For example, in a scenario where a mobile phone is outsourcing computation to a cloud, PartialGC can save the encrypted intermediate outputs to the cloud instead of the phone (Figure 6.2). This allows the phones to communicate with each other by storing encrypted intermediate values in the cloud, which is more efficient than requiring them to directly participate in the saving of values, as required by earlier 2P-SFE systems. Our friend finder application, built for an Android device, reflects this usage model and allows multiple friends to share their intermediate values

in a cloud. Other friends use these saved values to check whether or not someone is in the same map cell as themselves without having to copy and send data.

By incorporating our optimizations, we give the following contributions:

1. *Reusable Encrypted Values* – We show how to reuse an encrypted value, using only garbled circuits, by mapping one garbled value into another.

2. *Reduced Runtime and Bandwidth* – We show how reusable encrypted values can be used in practice to reduce the execution time for a garbled-circuit computation; we get a 96% reduction in runtime and a 98% reduction in bandwidth over CMTB.

   Impressively, we can reduce the amount of bandwidth required by the mobile party *arbitrarily* when no input checks have to be performed on the partial (intermediate) inputs in our protocol.

3. *Outsourcing Stateful Applications and Client-Server Communication Pattern* – We show how our system increases the scope of SFE applications by allowing multiple evaluating parties over a period of time to operate on the saved state of an SFE computation without the need for these parties to know about each other. This also allows us to operate using a traditional client-server communication pattern common to most everyday applications as opposed to the many-to-many communications required for most privacy-preserving protocols.

The remainder of this chapter is organized as follows: Section 4.2 introduces the concept of partial garbled circuits in detail. The PartialGC protocol and its implementation are described in Section 4.3, while its security is analyzed in Section 4.4. Section 4.5 evaluates PartialGC and introduces the friend finder application. Finally, Section 4.6 discusses related work.

The material in this chapter appeared in preliminary form in [20].

4.2 Partial Garbled Circuits

We introduce the concept of *partial garbled circuits* (PGCs), which allows the encrypted wire outputs from one SFE computation to be used as inputs to another. This can be accomplished by *mapping* the encrypted output wire values to valid input wire values in the next computation. In order to better demonstrate their structure and use, we first present PGCs in a semi-honest setting, before showing how they can aid us against malicious adversaries.

4.2.1 PGCs in the Semi-Honest Model

In the semi-honest model, for each wire value, the generator can simply send two values to the evaluator, which transforms the wire label the evaluator owns to work in another garbled circuit. Depending on the point and permute bit of the wire label received by the evaluator, she can map the value from a previous garbled circuit computation to a valid wire label in the next computation.

Specifically, for a given wire pair, the generator has wires w_0^{t-1} and w_1^{t-1}, and creates wires w_0^t and w_1^t. Here, t refers to a particular computation in a series, while 0 and 1 correspond to the values of the point and permute bits of the $t-1$ values. The generator sends the values $w_0^{t-1} \oplus w_0^t$ and $w_1^{t-1} \oplus w_1^t$ to the evaluator. Depending on the point and permute bit of the w_i^{t-1} value she possesses, the evaluator selects the correct value and then XORs her w_i^{t-1} with the $(w_i^{t-1} \oplus w_i^t)$ value, thereby giving her w_i^t, the valid partial input wire.

4.2.2 PGCs in the Malicious Model

In the malicious model we must allow the evaluation of a circuit with partial inputs and verification of the mappings, while preventing a selective failure attack. The following features are necessary to accomplish these goals:

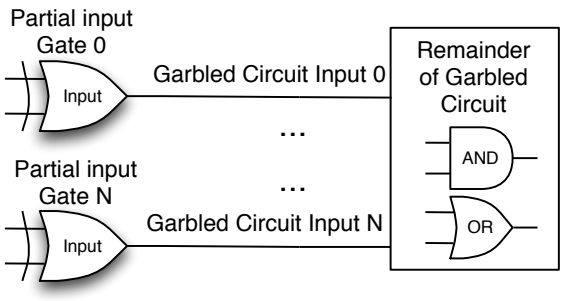


Figure 4.3: This figure shows how we create a single *partial input gate* for each input bit for each circuit and then link the *partial input gates* to the remainder of the circuit.

Verifiable Mapping: The generator G is able to create a secure mapping from a saved garbled wire value into a new computation that can be checked by the evaluator E, without E being able to reverse the mapping. During the evaluation and check phase, E must be able to verify the mapping G sent. G must have either committed to the mappings before deciding the partition of evaluation and check circuits, or never learned which circuits are in the check versus the evaluation sets.

Partial Generation and Partial Evaluation: G creates the garbled gates necessary for E to enter the previously output intermediate encrypted values into the next garbled circuit. These garbled gates are called *partial input gates*. As shown in Figure 4.3 each garbled circuit is made up of two pieces: the partial input gates and the remainder of the garbled circuit.

Revealing Incorrect Transformations: Our last goal is to let E inform G that incorrect values have been detected. Without a way to limit leakage, G could gain information based on whether or not E informs G that she caught him cheating. This is a selective failure attack and is not present in our protocol.

4.3 PartialGC Protocol

We start with the CMTB protocol and add cut-and-choose operations from sS13 before introducing the mechanisms needed to save and reuse values. We defer to

the original papers for full details of the outsourced oblivious transfer [36] and the generator's input consistency check [4] sub-protocols that we use as primitives in our protocol.

Our system operates in the same threat model as CMTB (see Section 4.4): we are secure against a malicious adversary under the assumption of non-collusion. A description of the CMTB protocol is available in Appendix A.1.

4.3.1 Preliminaries

There are three participants in the protocol:

Generator – As in standard garbled circuits, the generator is the party that generates the garbled circuit for the 2P-SFE.

Evaluator – As in standard garbled circuits, the evaluator is the other party in the 2P-SFE; it outsources computation to the cloud.

Cloud – We introduce a third party, the cloud, which executes the garbled circuit outsourced by the evaluator.

Notation

C_i - The ith circuit.

$CKey_i$ - Circuit key used for the free XOR optimization [40]. The key is randomly generated and then used as the difference between the 0 and 1 wire labels for a circuit C_i.

$CSeed_i$ - This value is created by the generator's PRNG and is used to generate a particular circuit C_i.

$POut\#_{i,j}$ - The *partial output* values are the encrypted wire values output from an SFE computation. These are encrypted garbled circuit values that can be reused in another garbled circuit computation. # is replaced in our protocol description with either a 0, 1, or x, signifying whether it represents a 0, 1, or an unknown value (from

the cloud's point of view). i denotes the circuit the $POut$ value came from and j denotes the wire of the $POut_i$ circuit.

$PIn\#_{i,j}$ - The *partial input* values are the re-entered $POut$ values after they have been obfuscated to remove the circuit key from the previous computation. These values are input to the *partial input gates*. $\#$, i, and j, are the same as above.

$GIn\#_{i,j}$ - The *garbled circuit input* values are the results of the partial input gates and are input into the remaining garbled circuit, as shown in Figure 4.3. $\#$, i, and j, are the same as above.

Partial Input Gates - These are garbled gates that take in PIn values and output GIn values. Their purpose is to transform the PIn values into values that are under $CKey_i$ for the current circuit.

Algorithm 0: PartialComputation

Input : Circuit_File, Bit_Security, Number_of_Circuits, Inputs, Is_First_Execution
Output: Circuit File Output
`Cut_and_Choose(`*is_First_Execution*`)`
Eval_Garbled_Input ← `Evaluator_Input(`*Eval_Select_Bits, Possible_Eval_Input*`)`
`Generator_Input_Check(`*Gen_Input*`)`
Partial_Garbled_Input ← `Partial_Input(`$Partial\_Output_{time-1}$`)`
Garbled_Output, Partial_Output ← `Circuit_Execution(`*Garbled_Input (Gen, Eval, Partial)*`)`
`Circuit_Output(`*Garbled_Output*`)`
`Partial_Output(`*Partial_Output*`)`

4.3.2 Protocol

Each computation is self-contained; other than what is explicitly described as saved in the protocol, each value or property is only used for a single part of the computation (*i.e.* randomness is different across computations).

Common Inputs: The program circuit file, the bit level security K, the circuit level security (number of circuits) S, and encryption and commitment functions.

Private Inputs: The evaluator's input $evlInput$ and generator's input $genInput$.

Outputs: The evaluator and generator can both receive garbled circuit outputs.

Phase 1: Preparation and Cut-and-choose

Algorithm 1: Cut_and_Choose

```
Input  : is_First_Execution
if is_First_Execution then
    circuitSelection ← rand() // bit-vector of size S
N ← (2/5)S // Number of evaluation circuits
//Generator creates his garbled input and circuit seeds for each circuit
for i ← 0 to S do
    CSeed_i ← rand()
    garbledGenInput_i ← garble(genInput, rand())
    //generator creates or loads keys
    if is_First_Execution then
        checkKey_i ← rand()
        evlKey_i ← rand()
    else
        loadKeys();
        checkKey_i ← hash(loadedCheckKey_i)
        evlKey_i ← hash(loadedEvlKey_i)
    // encrypts using unique one-time XOR pads
    encSeedIn_i ← CSeed_i ⊕ evlKey_i
    encGarbledIn_i ← garbledGenInput_i ⊕ checkKey_i
if is_First_Execution then
    // generator offers input OR keys for each circuit seed
    selectedKeys ← OT(circuitSelection, {evlKey, checkKey})
else
    loadSelectedKeys()
for i ← 0 to S do
    genSendToEval(hash(checkKey_i), hash(evaluationKey_i))
for i ← 0 to S do
    cloudSendToEval(hash(selectedKey_i), isCheckCircuit_i)
// If all values match, the evaluator learns split, else abort.
for i ← 0 to S do
    j ← isCheckCircuit_i
    correct ← (recievedGen_{i,j} == recievedEvl_i)
    if !correct then
        abort()
```

Preparation:

The generator creates two seeds for each circuit $C_0 \dots C_{S-1}$, $CSeed_i = \{0,1\}^K$.

We prepare our circuits such that any output to the generator or evaluator is output under a one-time pad, encrypted inside of the circuit. That is we augment all circuits such that $out_{evl} = out_{evl} \oplus outputKey_{evl}$ and $out_{gen} = out_{gen} \oplus outKey_{gen}$, where out_{evl} and out_{gen} is the initial output.

The generator and evaluator's input is extended to include the corresponding $outputKey$ and a K-bit secret key for a MAC.

Using the same technique as CMTB for input encoding to split the evaluator's input in K bits, where $bit_{j,0} \oplus \dots \oplus bit_{j,K-1} = evlInput_j$ for the jth bit of the evaluator's

input. The generator then creates the possible evaluator's input for each circuit C_i. To create the evaluator's input, the generator creates a key $IKey_i = \{0,1\}^K$ for the ith circuit, and a set of seeds, $evlInputSeeds0_j = \{0,1\}^K$ and $evlInputSeeds1_j = \{0,1\}^K$, where for $0 <= j < len(evlInput)$. Two seeds are created for each bit, representing 0 and 1. The garbled input values are then created:

$$garbledInputEvl0_{ij} = hash(evlInputSeeds0_j, IKey_i)$$
$$garbledInputEvl1_{ij} = hash(evlInputSeeds1_j, IKey_i)$$

As with CMTB, the possible evaluator's inputs are permuted for each different circuit to prevent the cloud from understanding what the evaluator's input maps to. The generator commits to each input value so the cloud will be able to verify he did not swap values.

Cut-and-choose:

Unlike some other GC protocols we do not commit to the various circuits before we execute the cut-and-choose. We modify the cut-and-choose mechanism described in sS13 as we have an extra party involved in the computation. In this cut-and-choose, the cloud selects which circuits are evaluation circuits and which circuits are check circuits,

$$circuitSelection = \{0,1\}^S$$

where 0 is an evaluation circuit and 1 is a check circuit. N evaluation circuits and $S - N$ check circuits are selected (like sS13, we use $N = \frac{2}{5}S$). The generator does not learn the circuit selection.

The generator generates garbled versions of his input and circuit seeds for each circuit. He encrypts these values using unique one-time XOR pad key for each circuit.

He also encrypts the evaluator's possible input. For $0 \leq i < S$,

$$garbledGenInput_i = garbleInput(genInput)$$
$$checkKey_i = \{0,1\}^K$$
$$evlKey_i = \{0,1\}^K$$
$$encGarbledIn_i = garbledGenInput_i \oplus evlKey_i$$
$$encSeedIn_i = CSeed_i \oplus checkKey_i$$
$$encInputEvl = garbledInputEvl \oplus checkKey_i$$

where $garbleInput()$ takes in the input, and produces a vector of $\{0,1\}^K$ bit strings, one for each bit of the generator's input for a given C_i and $garbledInputEvl$ is the garbled input
$(garbledInputEvl0_{i,0}|| \ldots ||garbledInputEvl0_{i,len-1}$
$||garbledInputEvl1_{i,0} || \ldots ||garbledInputEvl1_{i,len-1})$ and len is the length of $evlInput$.

The cloud and generator perform an oblivious transfer where the generator offers up decryption keys for his input and decryption keys for the circuit seed and possible evaluator's input for each circuit. The cloud can select the key to decrypt the generator's input or the key to decrypt the circuit seed and possible evaluator's input for a circuit but not both.

$$selectedKeys = OT(circuitSelection, \{evlKey, checkKey\})$$

For each circuit, if the cloud selects the decryption key for the circuit seed and possible evaluator's input in the oblivious transfer, then the circuit is used as a check circuit. If the cloud selects the key for the generator's input then the circuit is used as an evaluation circuit.

The generator sends the encrypted garbled inputs and check circuit information for all circuits to the cloud. The cloud decrypts the information he can decrypt using its keys. Both the cloud and generator save the decryption keys so they can be used in future computations, which use saved values.

The evaluator must also learn the circuit split. The generator sends a hash of each possible encryption key the cloud could have selected to the evaluator for each circuit as an ordered pair. For $0 \leq i < S$,

$$genSend(hash(checkKey_i), hash(evaluationKey_i))$$

The cloud sends a hash of the value received to the evaluator for each circuit. The cloud also sends bits to indicate which circuits were selected as check or evaluation circuits to the evaluator. For $0 \leq i < S$,

$$cloudSend(hash(selectedKey_i), isCheckCircuit_i)$$

The evaluator compares the hash the cloud sent to one of the hashes the generator sent, which is selected by the circuit selection sent by the cloud. For $0 \leq i < S$,

$$j = isCheckCircuit_i$$
$$correct = (receivedGen_{i,j} == receivedEvl_i)$$

If all values match, the evaluator uses the $isCheckCircuit_i$ to learn the split between check and evaluator circuits. Otherwise, abort.

We only perform the cut-and-choose oblivious transfer for the initial computation. For any subsequent computations, the generator and evaluator hash the saved decryption keys and use those hashes as the new encryption and decryption keys. The circuit split selected by the cloud is saved and stays the same across computations.

At the conclusion of this step (1) the cloud has all the information to evaluate the evaluation circuits when they are sent by the generator, i.e. the generator's input for each evaluation circuit, (2) the cloud has all the information to validate the check circuits when the generator sends those over, *i.e.*, each circuit seed and the possible evaluator's input for the check circuits (3) the cloud and evaluator know the check and evaluation circuit split, (4) the generator does not know the circuit split.

Phase 2: Evaluator's Input and Oblivious Transfer

Algorithm 2: Evaluator_Input

```
Input  : Eval_Select_Bits, Possible_Eval_Input
Output: Eval_Garbled_Input
// cloud gets selected input wires // generator offers both possible input wire values for each input wire;
evaluator selects its input
outSeeds = BaseOOT(bitsEvl, possibleInputs).
// the generator sends unique IKey values for each circuit to the evaluator
for i ← 0 to S do
    genSendToEval(IKey_i)
// the evaluator sends IKey values for all evaluation circuits to the cloud
for i ← 0 to S do
    if !isCheckCircuit(i) then
        EvalSendToCloud(IKey_i)
// cloud uses this to learn appropriate inputs
for i ← 0 to S do
    for j ← 0 to len(evlInputs) do
        if !isCheckCircuit(i) then
            inputEvl_ij ← hash(IKeys_i, outSeeds_j)
return inputEvl
```

We use the base outsourced oblivious transfer (OOT) of CMTB. In CMTB's OOT, the evaluator enters in the inputs buts and the generator enters in both possible inputs. The evaluator and generator perform a single OT operation before extending it, using the Ishai OT extension, to all the input bits. After extending it across each input bit it is then extend across each garbled circuit using the same technique described in the algorithm. After the OOT is finished, the cloud has the selected input wire values, which represent the evaluator's input.

As with CMTB, which uses the results from a single OOT as seeds to create the evaluator's input for all circuits, the cloud in our system also uses seeds from a single base OT (called "BaseOOT" below) to generate the input for the evaluation circuits.

The cloud receives the seeds for each input bit selected by the evaluator.

$$outSeeds = BaseOOT(evlInput, evlInputSeeds).$$

where *outSeeds* are the seeds selected by the evaluator's input.

The generator sends the $IKey_i$ keys (from phase 1) to the evaluator for each circuit. The evaluator sends the keys for the evaluation circuits to the cloud. The cloud then uses these keys and the *outSeeds* to attain the evaluator's input. For $0 \leq i < S$, for $0 \leq j < len(evlInputs)$ where $!isCheckCircuit(i)$,

$$inputEvl_{ij} = hash(IKey_i, outSeeds_j)$$

Phase 3: Generator's Input Consistency Check

Algorithm 3: Generator_Input_Check

```
Input  : Generator_Input
// The cloud takes a hash of the generator's input or each evaluation circuit for i ← 0 to S do
    if isCheckCircuit(i) then
        t_i ← UHF(garbledGenInput_i)
//If a single hash is different then the cloud knows the generator tried to cheat.
correct ← ((t_0 == t_1)&(t_0 == t_2)&...&(t_0 == t_{N-1}))
if !correct then
    abort()
```

We use the input consistency check of sS13. In this check, a universal hash is used to prove consistency of the generator's input across each evaluation circuit (attained in phase 1). Simply put, if any hash is different in any of the evaluation circuits, we know the generator did not enter consistent input. More formally, a hash of the generator's input is taken for each circuit. For $0 < i < S$ where $!isCheckCircuit(i)$,

$$t_i = UHF(garbledGenInput_i, C_i)$$

The results of these universal hashes are compared. If a single hash is different then

the cloud knows the generator tried to cheat.

$$correct = ((t_0 == t_1)\&(t_0 == t_2)\& \ldots \&(t_0 == t_{N-1}))$$

Phase 4: Partial Input Gate Generation, Check, and Evaluation

Algorithm 4: Partial_Input

```
Input  : Partial_Output
Output: Partial_Garbled_Input
// Generation: the generator creates a partial input gate, which transforms a wire's saved values, POut0_{i,j}
and POut1_{i,j}, into values that can be used in the current garbled circuit execution, GIn0_{i,j} and GIn1_{i,j}.
for i ← 0 to S do
    R_i ← PRNG.random()
    for j ← 0 to len(savedWires) do
        t0 ← hash(POut0_{i,j} ⊕ R_i)
        t1 ← hash(POut1_{i,j} ⊕ R_i)
        PIn0_{i,j}, PIn1_{i,j} ← setPPBitGen(t0, t1)
        GIn0_{i,j} ← TT0_{i,j} ⊕ PIn0_{i,j}
        GIn1_{i,j} ← TT1_{i,j} ⊕ PIn1_{i,j}
        GenSendToCloud( Permute([TT0_{i,j}, TT1_{i,j}]), permute_bit_locations )
    GenSendToCloud(R_i)
// Check: The cloud checks the gates to make sure the generator didn't cheat
for i ← 0 to S do
    if isCheckCircuit(i) then
        for j ← 0 to len(savedWires) do
            // the cloud has received the truth table information, TT0_{i,j}, TT1_{i,j}, bit locations from
            setPPBitGen, and R_i
            correct ← (generateGateFromInfo() == receivedGateFromGen())
            // If any gate does not match, the cloud knows the generator tried to cheat.
            if !correct then
                abort();
// Evaluation
for i ← 0 to S do
    if !isCheckCircuit(i) then
        for j ← 0 to len(savedWires) do
            //The cloud, using the previously saved POutx_{i,j} value, and the location (point and permute)
            bit sent by the generator, creates PInx_{i,j}
            PInx_{i,j} ← setPPBitEval(hash(R_i ⊕ POutx_{i,j}), location)
            // Using PInx_{i,j}, the cloud selects the proper truth table entry TTx_{i,j} from either TT0_{i,j} or
            TT1_{i,j} to decrypt
            // Creates GInx_{i,j} to enter into the garbled circuit
            GInx_{i,j} ← TTx_{i,j} ⊕ POutx_{i,j}
return GIn;
```

Generation:

For $0 \leq i < S$, for $0 \leq j < len(savedWires)$ the generator creates a *partial input gate*, which transforms a wire's saved values, $POut0_{i,j}$ and $POut1_{i,j}$, into values that can be used in the current garbled circuit execution, $GIn0_{i,j}$ and $GIn1_{i,j}$. For each circuit $0 \leq i < S$, the generator creates a pseudorandom transformation value

$R_i = \{0,1\}^K$, to assist with the transformation.

For each set of $POut0_{i,j}$ and $POut1_{i,j}$, the generator XORs each value with R_i. Both results are then hashed, and put through a function to determine the new permutation bit, as hashing removes the old permutation bit.

$$t0 = hash(POut0_{i,j} \oplus R_i)$$
$$t1 = hash(POut1_{i,j} \oplus R_i)$$
$$PIn0_{i,j}, PIn1_{i,j} = setPPBitGen(t0, t1)$$

This function, *setPPBitGen*, pseudo-randomly finds a bit that is different between the two values of the wire and notes that bit to be the permutation bit. *setPPBitGen* is seeded from $CSeed_i$, allowing the cloud to regenerate these values for the check circuits.

For each $PIn0_{i,j}, PIn1_{i,j}$ pair, a set of values, $GIn0_{i,j}$ and $GIn1_{i,j}$, are created under the master key of C_i – where $CKey_i$ is the difference between 0 and 1 wire labels for the circuit. In classic garbled gate style, two truth table values, $TT0_{i,j}$ and $TT1_{i,j}$, are created such that:

$$TT0_{i,j} \oplus PIn0_{i,j} = GIn0_{i,j}$$
$$TT1_{i,j} \oplus PIn1_{i,j} = GIn1_{i,j}$$

The truth table, $TT0_{i,j}$ and $TT1_{i,j}$, is permuted so that the permutation bits of $PIn0_{i,j}$ and $PIn1_{i,j}$ tell the cloud which entry to select. Each *partial input gate*, consisting of the permuted $TT0_{i,j}, TT1_{i,j}$ values, the bit location from $setPPBitGen$, and each R_i, is sent to the cloud.

Check:

For all the check circuits, (*i.e.*, $0 \leq i < S$ where $isCheckCircuit(i)$ is true), for $0 \leq$

$j < len(savedWires)$, the cloud receives the truth table information, $TT0_{i,j}, TT1_{i,j}$, and bit location from $setPPBitGen$, and proceeds to regenerate the gates based on the check circuit information. The cloud uses R_i (sent by the generator), $POut0_{i,j}$ and $POut1_{i,j}$ (saved during the previous execution), and $CSeed_i$ (recovered during the cut-and-choose) to generate the *partial input gates* in the same manner as described previously. The cloud then compares these gates to those the generator sent. If any gate does not match, the cloud knows the generator tried to cheat.

Evaluation:

For $0 \leq i < S$ where $!isCheckCircuit(i)$, for $0 \leq j < len(savedWires)$ the cloud receives the truth table information, $TTa_{i,j}, TTb_{i,j}$ and bit location from $setPPBitGen$. a and b are used to denote the two permuted truth table values. The cloud, using the previously saved $POutx_{i,j}$ value, creates the $PInx_{i,j}$ value

$$PInx_{i,j} = setPPBitEval(hash(R_i \oplus POutx_{i,j}), location)$$

where *location* is the location of the point and permute bit sent by the generator. Using the point and permute bit of $PInx_{i,j}$, the cloud selects the proper truth table entry $TTx_{i,j}$ from either $TTa_{i,j}$ or $TTb_{i,j}$ to decrypt, creates $GInx_{i,j}$ and then enters $GInx_{i,j}$ into the garbled circuit.

$$GInx_{i,j} = TTx_{i,j} \oplus POutx_{i,j}$$

Phase 5: Circuit Generation and Evaluation

Circuit Generation:

The generator generates every garbled gate for each circuit and sends them to the cloud. Since the generator does not know the check and evaluation circuit split, nothing changes between the generation for check and evaluation circuits. For $0 \leq$

Algorithm 5: Circuit_Execution

```
Input  : Generator_Input, Evaluator_Input, Partial_Input
Output: Partial_Output, Garbled_Output
// The generator generates each garbled gate and sends it to the cloud. Depending on whether the circuit is
a check or evaluation circuit, the cloud verifies that the gate is correct or evaluates the gate.
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← genGate(C_i, j)
        send(g)
// the cloud receives all gates for all circuits, and then checks OR evaluates each circuit
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← recvGate()
        if isCheckCircuit(i) then
            if ! verifyCorrect(g) then
                abort()
        else
            eval(g)
return Partial_Output, Garbled_Output
```

$i < S$, For $0 \leq j < len(circuit)$,

$$g = garbleGate(C_i, j)$$
$$send(g)$$

Circuit Evaluation and Check:

The cloud receives garbled gates for all circuits. For evaluation circuits the cloud evaluates those garbled gates. For check circuits the cloud generates the correct gate, based on the circuit seed, and is able to verify it is correct. For $0 \leq i < S$, For $0 \leq j < len(circuit)$,

$$g = recvGate()$$
$$if(isCheckCircuit(i)) \quad verifyCorrect(g)$$
$$else \quad eval(g)$$

If a garbled gate is found not to be correct, the cloud informs the evaluator and generator of the incorrect gate and safely aborts.

Phase 6: Output and Output Consistency Check

Algorithm 6: Circuit_Output

Input : Garbled_Output
// a MAC of the output is generated inside the garbled circuit, and both the resulting garbled circuit output and the MAC are encrypted under a one-time pad.
$outEvlComplete = outEvl||MAC(outEvl)$
$result = (outEvlMAC == MAC(outEvl))$
if *!result* **then**
  abort() // output check fail

As the final step of the garbled circuit execution, a MAC of the output is generated inside the garbled circuit, based on a k-bit secret key entered into the function.

$$outEvlComplete = outEvl||MAC(outEvl)$$

Both the resulting garbled circuit output and the MAC are encrypted under the one-time pad (from phase 1 before) leaving the garbled circuit.

To receive output from the garbled circuit for any particular output bit x, a majority vote is taken across all evaluation circuits. For $0 \leq i < S$ where $!isCheckCircuit(i)$,

$$result = majority(COut_{0,x} \ldots COut_{i-1,x})$$

Where $COut_{i,j}$ is the output bits, i is the ith circuit and j is the jth output bit from circuit i.

The cloud sends the corresponding encrypted (under the one-time pad introduced in phase 1) output to each party.

The generator and evaluator then decrypt the received ciphertext by using their one-time pad keys and perform a MAC over real output to verify the cloud did not modify the output by comparing the generated MAC with the MAC calculated within the garbled circuit.

$$result = (outEvlMAC == MAC(outEvl))$$

Both parties, the generator and evaluator, now have their output.

Phase 7: Partial Output

Algorithm 7: Partial_Output

```
Input  : Partial_Output
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        //The generator saves both possible wire values
        GenSave(Partial_Output0_{i,j})
        GenSave(Partial_Output1_{i,j})
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        if isCheckCircuit(i) then
            EvlSave(Partial_Output0_{i,j})
            EvlSave(Partial_Output1_{i,j})
        else
            // circuit is evaluation circuit EvlSave(Partial_OutputX_{i,j})
```

The generator saves both possible wire values for each partial output wire. For each evaluation circuit the cloud saves the partial output wire value. For check circuits the cloud saves both possible output values.

4.3.3 Implementation

As with most garbled circuit systems there are two stages to our implementation. The first stage is a compiler for creating garbled circuits, while the second stage is an execution system to evaluate the circuits.

We modified the compiler from Kreuter et al. [3] (hereon KSS12 compiler) to allow for the saving of intermediate wire labels and loading wire labels from a different SFE computation. By using the KSS12 compiler, we have an added benefit of being able to compare circuits of almost identical size and functionality between our system and CMTB, whereas other protocols compare circuits of sometimes vastly different sizes.

For our execution system, we started with the CMTB system and modified it according to our protocol requirements. PartialGC automatically performs the output consistency check, and we implemented this check at the circuit level. We became aware and corrected issues with CMTB relating to too many primitive OT operations

($S * inputs$ instead $inputs$) performed in the outsourced oblivious transfer when using a high circuit parameter and too low a general security parameter ($log_2(input)$ instead of 80). The fixes reduced the run-time of the OOT, though the exact amount varied.

4.4 Security of PartialGC

In this section, we provide a proof of the PartialGC protocol, showing that our protocol preserves the standard security guarantees provided by traditional garbled circuits - that is, none of the parties learns anything about the private inputs of the other parties that is not logically implied by the output it receives. That is, we show that our outsourced 2P-SFE protocol computes a function $f(x_1, x_2)$ securely (as described in Definition 1 for $n = 2$ parties) in the presence of a third party, the cloud, which does not learn any of the inputs or outputs, and does not collude with any other party. More detailed defintions are available in Kamara *et al.* [35] and also in Appendix A.4.

Section 4.4.1 provides a high-level overview of the proof. Section 4.4.2 goes over models and definitions, followed by security guarantees in Section 4.4.3; a full proof is provided in Appendix A.4.

Note on Non-collusion. CMTB assumes non-collusion, as quoted below:

"The outsourced two-party SFE protocol securely computes a function f(a,b) in the following two corruption scenarios: (1)The cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, (2)All but one party is malicious, while the cloud is semi-honest."

This is the standard definition of non-collusion used in server-aided works such as Kamara *et al.* [35]. Non-collusion does not mean the parties are trusted; it only means the two parties are not working together in order to cheat. In CMTB, any individual party that attempts to cheat to gain additional information will still be caught, but collusion between multiple parties could leak information. For instance,

the generator could send the cloud the keys to decrypt the circuit and see what the intermediate values are of the garbled function.

4.4.1 Proof Overview

We know that the protocol described in CMTB allows us to garble individual circuits and securely outsource their evaluation. Here, we modify certain portions of the protocol to allow us to transform the output wire values from a previous circuit execution into input wire values in a new circuit execution. These transformed values, which can be checked by the evaluator, are created by the generator using circuit "seeds."

We also use some aspects of sS13, notably their novel cut-and-choose technique which ensures that the generator does not learn which circuits are used for evaluation and which are used for checking - this means that the generator must create the correct transformation values for all of the cut-and-choose circuits.

Because we assume that the CMTB garbled circuit scheme can securely garble any circuit, we can use it individually on the circuit used in the first execution and on the circuits used in subsequent executions. We focus on the changes made at the end of the first execution and the beginning of subsequent executions which are introduced by PartialGC.

The only difference between the initial garbled circuit execution and any other garbled circuit in CMTB is that the output wires in an initial PartialGC circuit are stored by the cloud, and are not delivered to the generator or the evaluator. This prevents them from learning the output wire labels of the initial circuit, but cannot be less secure than CMTB, since no additional steps are taken here.

Subsequent circuits we wish to garble differ from ordinary CMTB garbled circuits only by the addition, before the first row of gates, of a set of partial input gates. These gates don't change the output along a wire, but differ from normal garbled

gates in that the two possible labels for each input wire are not chosen randomly by the generator, but are derived by using the two labels along each output wire of the initial garbled circuit.

This does not reduce security. In PartialGC, the input labels for partial input gates have the same property as the labels for ordinary garbled input gates: the generator knows both labels, but does not know which one corresponds to the evaluator's input, and the evaluator knows only the label corresponding to its input, but not the other label. This is because the evaluator's input is exactly the output of the initial garbled circuit, the output labels of which were saved by the evaluator. The evaluator does not learn the other output label for any of the output gates because the output of each garbled gate is encrypted. If the evaluator could learn any output labels other than those which result from an evaluation of the garbled circuit, the original garbled circuit scheme itself would not be secure.

The generator, which also generated the initial garbled circuit, knows both possible input labels for all partial evaluation gates, because it has saved both potential output labels of the initial circuit's output gates. Because of the outsourced oblivious transfer used in CMTB, the generator did not know which input labels to use for the initial garbled circuit, and therefore will not have been able to determine the output labels for that circuit. Therefore, the generator will likewise not know which input labels are being used for subsequent garbled circuits.

4.4.2 Model and definitions

Throughout our protocol, we assume that none of the parties involved ever collude with the cloud. It is known that theoretical limitations exist when considering collusion in secure multiparty computation, and other schemes considering secure computation with multiple parties require similar restrictions on who and how many parties may collude while preserving security. If an outsourcing protocol is secure

when both the generator and the cloud are malicious and colluding, this implies a secure two-party scheme where one party has sub-linear work with respect to the size of the circuit, which is currently only possible with fully homomorphic encryption [35]. However, making the assumption that the cloud will not collude with the participating parties makes outsourcing securely a theoretical possibility.

While it is unlikely that a reputable cloud provider would allow external parties to illegally control or modify computations within their systems, we cannot assume the cloud will automatically be semi-honest. For example, our protocol requires a number of consistency checks to be performed by the cloud that ensure the participants do not cheat. Without mechanisms to force the cloud to make these checks, a "lazy" cloud provider could save resources by simply returning that all checks verified without actually performing them.

The work of Kamara et al. [35] formalizes the idea of a non-colluding cloud based on the ideal-model/real-model security definitions common in secure multiparty computation. We apply their definitions to our protocol (for the two-party case in particular) as described below. These definitions are also used for the full proof presented in Appendix A.4.

Real-model execution. The protocol takes place between two parties (P_1, P_2) executing the protocol and a server P_3, where each of the executing parties provides input x_i, auxiliary input z_i, and random coins r_i. The server provides only auxiliary input z_3 and random coins r_3. There exists some subset of independent parties $(A_1, ..A_m), m \leq 3$ that are malicious adversaries. Each adversary corrupts one executing party and does not share information with other adversaries. For all honest parties, let OUT_i be its output, and for corrupted parties let OUT_i be its view of the protocol execution. The i^{th} partial output of a real execution is defined as $REAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$, where H is the set of honest parties and r is all random coins of all players.

Ideal-model execution. In the ideal model, the setup of participants is the same except that all parties are interacting with a trusted party that evaluates the function. All parties provide inputs x_i, auxiliary input z_i, and random coins r_i. If a party is semi-honest, it provides its actual inputs to the trusted party, while if the party is malicious or non-colluding, it provides arbitrary input values. In the case of the server P_3, this means simply providing its auxiliary input and random coins, as no input is provided to the function being evaluated. Once the function is evaluated by the trusted third party, it returns the result to the parties P_1 and P_2, while the server P_3 does not receive the output. If a party aborts early or sends no input, the trusted party immediately aborts. For all honest parties, let OUT_i be its output to the trusted party, and for corrupted parties let OUT_i be some value output by P_i. The i^{th} partial output of an ideal execution in the presence of some set of independent simulators is defined as $IDEAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \cup OUT_i$ where H is the set of honest parties and r is all random coins of all players.

Definition 2. *A protocol securely computes a function f if there exists a set of probabilistic polynomial-time (PPT) simulators* $\{Sim_i\}_{i\in[3]}$ *such that for all PPT adversaries* $(A_1, \ldots, A_3), x, z$, *and for all* $i \in [3]$, *we have*

$$\{REAL^{(i)}(k, x; r)\}_{k\in N} \approx \{IDEAL^{(i)}(k, x; r)\}_{k\in N}$$

Where $S = (S_1, \ldots, S_3), S_i = Sim_i(A_i)$, *and* r *is random and uniform.*

4.4.3 Security Guarantees

Generator's Input Consistency Check

During the cut-and-choose, multiple copies of the garbled circuit are constructed and then either checked or evaluated. A malicious generator may provide inconsistent inputs to different evaluation circuits. For some functions, it is possible to use inconsistent inputs to extract information of Eval's input [32].

Claim 1. *The generator in our protocol cannot trick the evaluator into using different inputs for different evaluation circuits with greater than negligible probability.*

We use the generator's input consistency check from [4], and defer to the proof provided in that paper, noting that simulators S_1 and S_2 can be constructed such that any malicious generator (resp. evaluator) cannot tell whether it is working with S_1 (resp. S_2) in the ideal model, or with an honest evaluator (resp. generator) in the real model.

We further note there is no problem with allowing the cloud to perform this check; for the generator's inconsistent input to pass the check, the cloud would have to see the malicious input and ignore it, which would violate the non-collusion assumption.

Validity of Evaluator Inputs

To ensure that the generator cannot learn anything about the evaluator's inputs by corrupting the garbled values sent during the OT, we use from CMTB the random input encoding technique by Lindell and Pinkas [32]. This technique allows the evaluator to encode each input bit as the XOR of a set of input bits. Thus, if the generator corrupts one of those input bits as in a selective failure attack, it reveals nothing about the evaluator's true input. Additionally, we use the commitment technique employed by Kreuter et al. [3] to ensure that the generator cannot swap garbled input wire labels between the zero and one value. To accomplish this, the generator commits to the wire labels before the cut and choose. During the cut and choose, the input labels for the check circuits are opened to ensure that they correspond to only one value across all circuits. Then, during the OOT, the commitment keys for the labels that will be evaluated are sent instead of the wire labels themselves. Because our protocol implements this technique directly from previous work, we do not make any additional claims of security.

Correctness of Saved Values

Scenarios where either party enters incorrect values in the next computation re-

duce to previously solved problems in garbled circuits. If the generator does not use the correct values, then it reduces to the problem of creating an incorrect garbled circuit. If the evaluator does not use the correct saved values then it reduces to the problem of the evaluator entering garbage values into the garbled circuit execution; this would be caught by the output consistency check.

Garbled Circuit Generation

To ensure the evaluated circuits are generated honestly, we require two properties. First, we limit the generator's ability to trick the evaluator into evaluating a corrupted circuit using a cut-and-choose technique similar to a typical, two-party garbled circuit evaluation. Second, we ensure that a lazy Cloud attempting to conserve system resources cannot bypass the circuit checking step without being discovered.

Claim 2. *Security: Assuming that the hash function $UHF(x)$ (as used in phase 3) is a one-way, collision-resistant hash and that the commitment scheme used is fully binding, then the generator has at best a 2^{-k} probability of tricking the evaluator into evaluating a majority of corrupted circuits, where k is the number of circuits generated.*

This claim follows directly from sS13. The probability of the generator finding a hash collision and thus fooling the evaluator is at most $1/|B|$, where B is the range of the hash function.

Claim 3. *Proof-of-work: Assuming the hash function is one-way and collision resistant, the Cloud has a negligible probability of producing a check hash that passes the seed check without actually generating the check circuit.*

As previously stated, before the circuit check begins the generator sends the evaluator k hashed circuit values $H_1(GC_i)$. Once the evaluation circuits are selected, the cloud must generate some circuits and hash them into check hashes $H_1(GC'_i)$. If the cloud attempts to skip the generation of the check circuits, it must generate hash

values $H'_i = H_i$ for $i \in Chk$. Based on security guarantees of the hash, and the non-collusion property, the cloud has a negligible probability of correctly generating these hash values.

Abort on Check Failure

If any of the check circuits fail, the cloud reports the incorrect check circuit to both the generator and evaluator. At this point, the remaining computation and any saved values must be abandoned. However, as is standard in SFE, the cloud cannot abort on an incorrect evaluation circuit even when it is known to be incorrect.

Concatenation of Incorrect Circuits

If the generator produces a single incorrect circuit and the cloud does not abort, the generator learns that the circuit was used for evaluation, and not as a check circuit. This leaks no information about the input or output of the computation; to do that, the generator must corrupt a majority of the evaluation circuits without modifying a check circuit. An incorrect circuit that goes undetected in one execution has no effect on subsequent executions as long the total amount of incorrect circuits is less than the majority of evaluation circuits.

Using Multiple Evaluators

One of the benefits of our outsourcing scheme is that the state is saved at the generator and cloud allowing the use of different evaluators in each computation. Previously, it was shown a group of users working with a single server using 2P-SFE was not secure against malicious adversaries, as a malicious server and last k parties, also malicious, could replay their portion of the computation with different inputs and gain more information than they can with a single computation [56]. However, this is not a problem in our system as at least one of our servers, either the generator or cloud, must be semi-honest due to non-collusion, which obviates the attack stated above.

Threat Model

As we have many computations involving the same generator and cloud, we have to extend the threat model for how the parties can act in different computations. There can be no collusion in each singular computation. However, the malicious party can change between computations as long as there is no chain of malicious users that link the generator and cloud – this would break the non-collusion assumption.

4.5 Performance Evaluation

We now demonstrate the efficacy of PartialGC through a comparison with the CMTB outsourcing system. Apart from the performance gains from using cut-and-choose from sS13, PartialGC provides other benefits through generating partial input values after the first execution of a program. On subsequent executions, the partial inputs act to amortize overall costs of execution and bandwidth.

We demonstrate that the evaluator in the system can be a mobile device outsourcing computation to a more powerful system. We also show that other devices, such as server-class machines, can act as evaluators, to show the generality of this system. Our testing environment includes a 64-core server containing 1 TB of RAM, which we use to model both the Generator and Outsourcing Proxy parties. We run separate programs for the Generator and Outsourcing Proxy, giving them each 32 threads. For the evaluator, we use a Samsung Galaxy Nexus phone with a 1.2 GHz dual-core ARM Cortex-A9 and 1 GB of RAM running Android 4.0, connected to the server through an 802.11 54 Mbps WiFi in an isolated environment. In our testing we also use a single server process as the evaluator. For these tests we create that process on our 64-core server as well. We ran the CMTB implementation for comparison tests under the same setup.

4.5.1 Execution Time

The PartialGC system is particularly well suited to complex computations that require multiple stages and the saving of intermediate state. Previous garbled circuit execution systems have focused on single-transaction evaluations, such as computing the "millionaires" problem (i.e., a joint evaluation of which party inputs a greater value without revealing the values of the inputs) or evaluating an AES circuit.

Our evaluation considers two comparisons: the improvement of our system compared with CMTB without reusing saved values, and comparing our protocol for saving and reusing values against CMTB if such reuse was implemented in that protocol. We also benchmark the overhead for saving and loading values on a per-bit basis for 256 circuits, a necessary number to achieve a security parameter of 2^{-80} in the malicious model. In all cases, we run 10 iterations of each test and give timing results with 95% confidence intervals. Other than varying the number of circuits our system parameters are set for 80-bit security.

The programs used for our evaluation are exemplars of differing input sizes and differing circuit complexities:

Keyed Database: In this program, one party enters a database and keys to it while the other party enters a key that indexes into the database, receiving a database entry for that key. This is an example of a program expressed as a small circuit that has a very large amount of input.

Matrix Multiplication: Here, both parties enter 32-bit numbers to fill a matrix. Matrix multiplication is performed before the resulting matrix is output to both parties. This is an example of a program with a large amount of inputs with a large circuit.

Edit (Levenstein) Distance: This program finds the distance between two strings of the same length and returns the difference. This is an example of a program with a small number of inputs and a medium sized circuit.

	CMTB	PartialGC
KeyedDB 64	6,080	20,891
KeyedDB 128	12,160	26,971
KeyedDB 256	24,320	39,131
MatrixMult8x8	3,060,802	3,305,113
Edit Distance 128	1,434,888	1,464,490
Millionaires 8192	49,153	78,775
LCS Incremental 128	4,053,870	87,236
LCS Incremental 256	8,077,676	160,322
LCS Incremental 512	16,125,291	306,368
LCS Full 128	2,978,854	-
LCS Full 256	13,177,739	-

Table 4.1: Non-XOR gate counts for the various circuits. In the first 6 circuits, the difference between CMTB and PartialGC gate counts is in the consistency checks. The explanation for the difference in size between the incremental versions of longest common substring (LCS) is given in *Reusing Values*.

Millionaires: In this classic SFE program, both parties enter a value, and the result is a one-bit output to each party to let them know whether their value is greater or smaller than that of the other party. This is an example of a small circuit with a large amount of input.

Gate counts for each of our programs can be found in Table 4.1. The only difference for the programs described above is the additional of a MAC function in PartialGC. We discuss the reason for this check in Section 4.5.4.

Table 4.2 shows the results from our experimental tests. In the best case, execution time was reduced by a factor of 32 over CMTB, from 1200 seconds to 38 seconds, a 96% speedup over CMTB. Ultimately, our results show that our system outperforms CMTB when the input checks are the bottleneck. This run-time improvement is due to improvements we added from sS13 and occurs in the keyed database, millionaires, and matrix multiplications programs. In the other program, edit distance, the input checks are not the bottleneck and PartialGC does not outperform CMTB. The total run-time increase for the edit distance problem is due to overhead of using the new sS13 OT cut-and-choose technique which requires sending each gate to the evaluator

	16 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	18 ± 2%	3.5 ± 3%	5.1x
KeyedDB 128	33 ± 2%	4.4 ± 8%	7.5x
KeyedDB 256	65 ± 2%	4.6 ± 2%	14x
MatrixMult8x8	48 ± 4%	46 ± 4%	1.0x
Edit Distance 128	21 ± 6%	22 ± 3%	0.95x
Millionaires 8192	35 ± 3%	7.3 ± 6%	4.8x
	64 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	72 ± 2%	8.3 ± 5%	8.7x
KeyedDB 128	140 ± 2%	9.5 ± 4%	15x
KeyedDB 256	270 ± 1%	12 ± 6%	23x
MatrixMult8x8	110 ± 8%	100 ± 7%	1.1x
Edit Distance 128	47 ± 7%	50 ± 9%	0.94x
Millionaires 8192	140 ± 2%	20 ± 2%	7.0x
	256 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	290 ± 2%	26 ± 2%	11x
KeyedDB 128	580 ± 2%	31 ± 3%	19x
KeyedDB 256	1200 ± 3%	38 ± 5%	32x
MatrixMult8x8	400 ± 10%	370 ± 5%	1.1x
Edit Distance 128	120 ± 9%	180 ± 6%	0.67x
Millionaires 8192	580 ± 1%	70 ± 2%	8.3x

Table 4.2: Timing results comparing PartialGC to CMTB without saving any values. All times in seconds.

for check circuits and evaluation circuits. This is discussed further in Section 4.5.4. The typical use case we imagine for our system, however, is more like the keyed database program, which has a large amount of inputs and a very small circuit. We expand upon this use case later in this section.

Reusing Values

For a test of our system's wire saving capabilities we tested a dynamic programming problem, longest common substring, in both PartialGC and CMTB. This program determines the length of the longest common substring between two strings. Rather than use a single computation for the solution, our version incrementally adds a single bit of input to both strings each time the computation is run and outputs

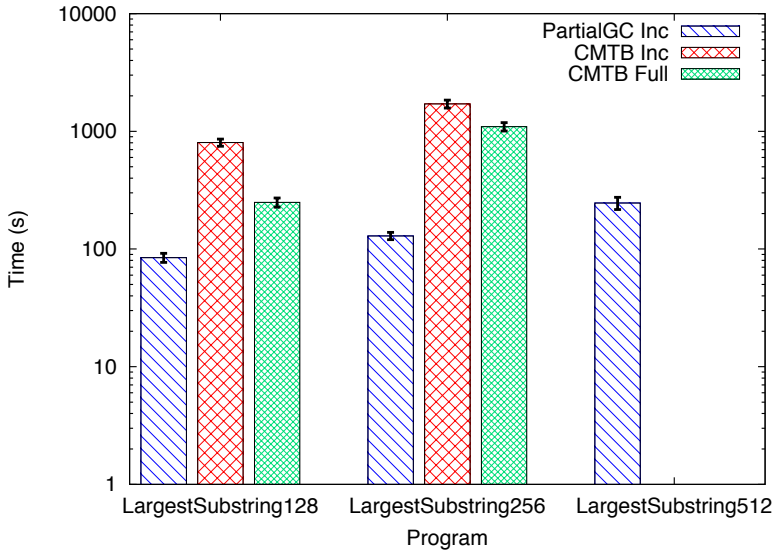


Figure 4.4: Results from testing our largest common substring (LCS) programs for PartialGC and CMTB. This shows when changing a single input value is more efficient under PartialGC than either CMTB program. CMTB crashed on running LCS Incremental of size 512 due to memory requirements. We were unable to complete the compilation of CMTB Full of size 512.

the results each time to the evaluator. We believe this is a realistic comparison to a real-world application that incrementally adds data during each computation where it is faster to save the intermediate state and add to it after seeing an intermediate result than rerun the entire computation many times after seeing the result.

For our testing, PartialGC uses our technique to reuse wire values. In CMTB, we save each desired internal bit under a one-time pad and re-enter them into the next computation, as well as the information needed to decrypt the ciphertext. We use a MAC (the AES circuit of KSS12) to verify that the party saving the output bits did not modify them. We also use AES to generate a one-time pad inside the garbled circuit. We use AES as this is the only cryptographically secure function used in CMTB. Both parties enter private keys to the MAC functions. This program is labeled *CMTB-Inc*, for CMTB *incremental*. The size of this program represents the size of the total strings. We also created a circuit that computes the complete longest common substring in one computation labeled *CMTB-Full*.

The resulting size of the PartialGC and CMTB circuits are shown in Table 4.1, and the results are shown in Figure 4.4. This result shows that saving and reusing values in PartialGC is more efficient than completely rerunning the computation. The input consistency check adds considerably to the memory use on the phone for *CMTB-Inc* and in the case of input bit 512, the *CMTB-Inc* program will not complete. In the case of the 512-bit *CMTB-Full*, the program would not complete compilation in over 42 hours. In our *CMTB-Inc* program, we assume the cloud saves the output bits so that multiple phones can have a shared private key.

Note that the growth of *CMTB-Inc* and *CMTB-Full* are different. *CMTB-Full* grows at a larger rate (4x for each 2x factor increase) than *CMTB-Inc* (2x for each 2x factor increase), implying that although at first it seems more efficient to rerun the program if small changes are desired in the input, eventually this will not be the case. Even with a more efficient AES function, *CMTB-Inc* would not be faster as the bottleneck is the input, not the size of the circuit.

The overhead of saving and reusing values is discussed further in Appendix A.2.

Outsourcing to a Server Process

PartialGC can be used in other scenarios than just outsourcing to a mobile device. It can outsource garbled circuit evaluation from a single server process and retain performance benefits over a single server process of CMTB. For this experiment the outsourcing party has a single thread. Table 4.3 displays these results and shows that in the KeyedDB 256 program, PartialGC has a 92% speedup over CMTB. As with the outsourced mobile case, keyed database problems perform particularly well in PartialGC. Because the computationally-intensive input consistency check is a greater bottleneck on mobile devices than servers, these improvements for most programs are less dramatic. In particular, both edit distance and matrix multiplication programs benefit from higher computational power and their bottlenecks on a server are no longer input consistency; as a result, they execute faster in CMTB than in PartialGC.

	16 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	6.6 ± 4%	1.4 ± 1%	4.7x
KeyedDB 128	13 ± 3%	1.8 ± 2%	7.2x
KeyedDB 256	25 ± 4%	2.5 ± 1%	10x
MatrixMult8x8	42 ± 3%	41 ± 4%	1.0x
Edit Distance 128	18 ± 3%	18 ± 3%	1.0x
Millionaires 8192	13 ± 4%	3.2 ± 1%	4.1x
	64 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	27 ± 4%	5.1 ± 2%	5.3x
KeyedDB 128	54 ± 4%	5.8 ± 2%	9.3x
KeyedDB 256	110 ± 7%	7.3 ± 2%	15x
MatrixMult8x8	94 ± 4%	79 ± 3%	1.2x
Edit Distance 128	40 ± 8%	40 ± 6%	1.0x
Millionaires 8192	52 ± 3%	8.5 ± 2%	6.1x
	256 Circuits		
	CMTB	PartialGC	Improvement
KeyedDB 64	110 ± 2%	24.9 ± 0.3%	4.4x
KeyedDB 128	220 ± 5%	27.9 ± 0.5%	7.9x
KeyedDB 256	420 ± 4%	33.5 ± 0.6%	13x
MatrixMult8x8	300 ± 10%	310 ± 1%	0.97x
Edit Distance 128	120 ± 9%	150 ± 3%	0.8x
Millionaires 8192	220 ± 5%	38.4 ± 0.9%	5.7x

Table 4.3: Timing results from outsourcing the garbled circuit evaluation from a single server process. Results in seconds.

4.5.2 Bandwidth

Since the main reason for outsourcing a computation is to save on resources, we give results showing a decrease in the evaluator's bandwidth. Bandwidth is counted by making the evaluator to count the number of bytes PartialGC sends and receives to either server. Our best result gives a 98% reduction in bandwidth (see Table 4.4). For the edit distance, the extra bandwidth used in the outsourced oblivious transfer for all circuits, instead of only the evaluation circuits, exceeds any benefit we would otherwise have received.

	256 Circuits		
	CMTB	PartialGC	
KeyedDB 64	64992308	3590416	18x
KeyedDB 128	129744948	3590416	36x
KeyedDB 256	259250228	3590416	72x
MatrixMult8x8	71238860	35027980	2.0x
Edit Distance 128	2615651	4108045	0.64x
Millionaires 8192	155377267	67071757	2.3x

Table 4.4: Bandwidth comparison of CMTB and PartialGC. Bandwidth counted by instrumenting PartialGC to count the bytes it was sending and receiving and then adding them together. Results in bytes.

4.5.3 Secure Friend Finder

Many privacy-preserving applications can benefit from using PartialGC to cache values for state. As a case study, we developed a privacy-preserving friend finder application, where users can locate nearby friends without any user divulging their exact location. In this application, many different mobile phone clients use a consistent generator (a server application) and outsource computation to a cloud. The generator must be the same for all computations; the cloud must be the same for each computation. The cloud and generator are two different parties. After each computation, the map is updated when PartialGC saves the current state of the map as wire labels. Without PartialGC outsourcing values to the cloud, the wire labels would have to be transferred directly between mobile devices, making a multi-user application difficult or impossible.

We define three privacy-preserving operations that comprise the application's functionality:

MapStart - The three parties (generator, evaluator, cloud) create a "blank" map region, where all locations in the map are blank and remain that way until some mobile party sets a location to his or her ID.

MapSet - The mobile party sets a single map cell to a new value. This program

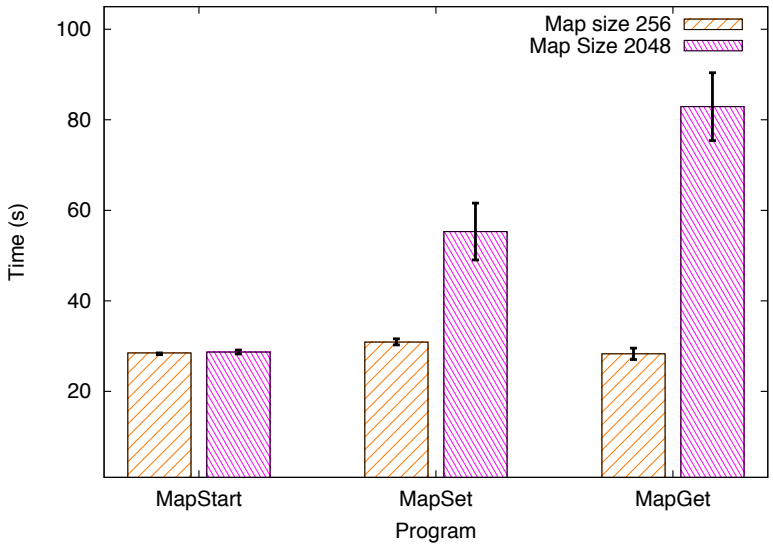


Figure 4.5: Run time comparison of our map programs with two different map sizes.

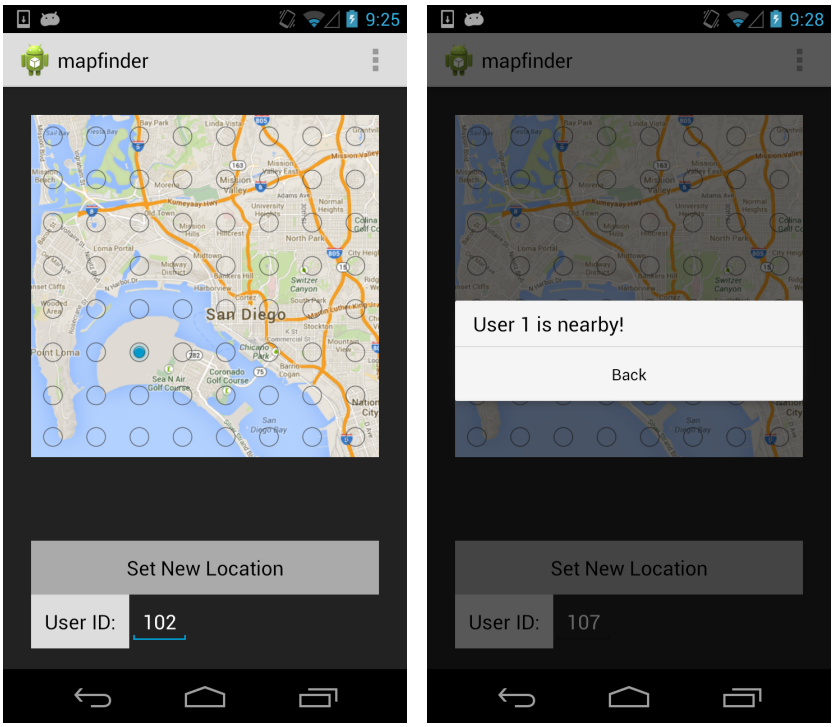
takes in partial values from the generator and cloud and outputs a location selected by the mobile party.

MapGet - The mobile party retrieves the contents of a single map cell. This program retrieves partial values from the generator and cloud and outputs any ID set for that cell to the mobile.

In the application, each user using the *Secure Friend Finder* has a unique ID that represents them on the map. We divide the map into "cells", where each cell is a set amount of area. When the user presses "Set New Location," the program will first look to determine if that cell is occupied. If the cell is occupied, the user is informed he is near a friend. Otherwise the cell is updated to contain his user ID and remove his ID from his previous location. We assume a maximum of 255 friends in our application since each cell in the map is 8 bits.

Figure 4.5 shows the performance of these programs in the malicious model with a 2^{-80} security parameter (evaluated over 256 circuits). We consider map regions containing both 256 and 2048 cells. For maps of 256 cells, each operation takes about 30 seconds.[1] As there are three operations for each "Set New Location" event, the

1. Our 64-cell map, as seen in figure 4.5, also takes about 30 seconds for each operation.



(a) Location selected. (b) After computation.

Figure 4.6: Screenshots from our application. (a) shows the map with radio buttons a user can select to indicate position. (b) show the result after "set new position" is pressed when a user is present. The application is set to use 64 different map locations. Map image from Google Maps.

total execution time is about 90 seconds, while execution time for 2048 cells is about 3 minutes. The bottleneck of the 64 and 256 cell maps is the outsourced oblivious transfer, which is not affected by the number of cells in the map. The vastly larger circuit associated with the 2048-cell map makes getting and setting values slower operations, but these results show such an application is practical for many scenarios.

Example - As an example, two friends initiate a friend finder computation using Amazon as the cloud and Facebook as the generator. The first friend goes out for a coffee at a café. The second friend, riding his bike, gets a message that his friend is nearby and looks for a few minutes and finds him in the café. Using this application prevents either Amazon or Facebook from knowing either user's location while they are able to learn whether they are nearby.

4.5.4 Discussion

Analysis of improvements

We analyzed our results and found the improvements came from three places: the improved sS13 consistency check, the saving and reusing of values, and the fixed oblivious transfer. In the case of the sS13 consistency check, there are two reasons for the improvement: first, there is less network traffic, and second, it uses symmetric key operations instead of exponentiations. In the case of saving and reusing values, we save time with the faster input consistency check and by not requiring a user to recompute a circuit multiple times. Lastly, we reduced the runtime and bandwidth by fixing parts of the OOT. The previous outsourced oblivious transfer performed the primitive OT S (S being the number of circuits) times instead of a single time, which turn forced many extra exponentiations. Each amount of improvement varies depending upon the circuit.

Output check

Although the garbled circuit is larger for our output check, this check performs less cryptographic operations for the outsourcing party, as the evaluator only has to perform a MAC on the output of the garbled circuit. We use this check to demonstrate using a MAC can be an efficient output check for a low power device when the computational power is not equivalent across all parties.

Commit Cut-and-Choose vs OT Cut-and-Choose

Our results unexpectedly showed that the sS13 OT cut-and-choose used in PartialGC is actually slower than the KSS12 commit cut-and-choose used in CMTB in our experimental setup. Theoretically, sS13, which requires fewer cryptographic operations, as it generates the garbled circuit only once, should be the faster protocol. The difference between the two cut-and-choose protocols is the network usage – instead of $\frac{2}{5}$ of the circuits (CMTB), *all* the circuits must be transmitted in sS13. The sS13 cut-and-choose is required in our protocol so that the cloud can check that the

generator creates the correct gates.

4.5.5 Implementation Optimizations

We proceeded to optimize our system in light of the slowdown we saw when compared to CMTB for circuits with large amounts of gates. We made the following changes: (1) turn AES-NI on, as it was not turned on by default in CMTB (or [3], which CMTB is based on), (2) hand-optimize the garbled gate generation and evaluation to remove excess memory operations, (3) remove the need for network I/O for XOR gates from the underlying implementation (previously, 4 bytes were spuriously transmitted for each XOR gate), (4) batch process gates to reduce the overhead of networking for each gate, and (5) remove unnecessary hash calls that existed in PartialGC as an artifact of being built on CMTB.

4.5.6 Corrections of Underlying Implementation

We made two corrections to the implementation of PartialGC that are artifacts of the underlying implementations. The first error was from KSS and while the other was from CMTB. We performed the following changes: (1) further correct the OT phase of CMTB and (2) add a missing input encoding phase that was supposed to exist in KSS. The first error was straightforward: rather than performing a single set of OTs and then extending it to all circuits, after the single set of OTs in CMTB, a matrix transformation was performed for each circuit (instead of a single matrix transformation). We removed this error and added the necessary correction, i.e. after the single set of OTs were performed, the results from the OTs were extended in the same manner as in our protocol description. To correct the second error, we added the missing input encoding step for the evaluator's input. Note that KSS and all subsequent systems built from it do not have this input encoding. Without the input encoding, a selective failure attack can be performed easily by the generator in order

	16 Circuits		
	New	Original	Improvement
KeyedDB 64	4 ± 10%	3.5 ± 3%	0.92x
KeyedDB 128	3.8 ± 10%	4.4 ± 8%	1.1x
KeyedDB 256	4.0 ± 4%	4.6 ± 2%	1.1x
MatrixMult8x8	21 ± 2%	46 ± 4%	2.2x
EditDist 128	7.8 ± 4%	22 ± 3%	2.8x
Millionaires 8192	24 ± 5%	7.3 ± 6%	0.30x
	64 Circuits		
	New	Original	Improvement
KeyedDB 64	4.4 ± 5%	8.3 ± 5%	1.9x
KeyedDB 128	4.5 ± 8%	9.5 ± 4%	2.1x
KeyedDB 256	4.7 ± 9%	12 ± 6%	2.7x
MatrixMult8x8	29 ± 4%	100 ± 7%	3.5x
EditDist 128	10 ± 4%	50 ± 9%	4.8x
Millionaires 8192	30 ± 3%	20 ± 2%	0.68x
	256 Circuits		
	New	Original	Improvement
KeyedDB 64	7.6 ± 6%	26 ± 2%	3.4x
KeyedDB 128	8.1 ± 4%	31 ± 3%	3.8x
KeyedDB 256	9.3 ± 4%	38 ± 5%	4.0x
MatrixMult8x8	69 ± 2%	370 ± 5%	5.4x
EditDist 128	21 ± 2%	180 ± 6%	8.9x
Millionaires 8192	78 ± 3%	70 ± 2%	0.89x

Table 4.5: Comparing the original PartialGC and the improved version of PartialGC. Results in seconds.

to gain information about a single bit of the evaluator‘s input.

4.5.7 Results from Optimal Implementation

In Table 4.5 we present results from the corrected and more optimal implementation of PartialGC. We observe the following: (1) the program that has a large evaluator's input and very little gates is slightly slower due to the fixed OT error and added input encoding (Millionaires). (2) The program with a large circuit size when compared with the input sizes of both the generator and evaluator has improved runtime performance (Edit distance). (3) The program we tested that has high input and also has a high gate count is improved (Matrix Mult). (4) The program that relies mostly on the

generator's input size with a low amount of gates is largely unaffected by the OT change or the added input encoding but is still improved by the optimizations to the garbled gate runtime (Keyed DB).

4.5.8 SFE Engineering Insights

Given our experience from building on other frameworks, we provide our insights:

1. We found that if the running time is not as expected then there is most likely something incorrect in the implementation, rather than an error in the application code. The latter usually causes a complete failure in our experience. For instance, we found that if the average time to evaluate garbled gates is greater than the average time to generate the garbled gates there is most likely a problem in the garbled circuit evaluation phase.
2. Although comparing the time of garbling and evaluating can be interesting in its own right, evaluating the total time of full garbled circuit garbling and evaluation (including network overhead) is also insightful as networking and related operations can be the bottleneck in a practical system. This includes network usage, the effects of a cut-and-choose protocol, and the time it takes to get the next gate from the interpreter or circuit file.
3. When using frameworks or compilers written by other people, check to verify each protocol step exists in the implementation. We found that a lot of systems skip vital parts of their (stated) protocols.
4. Implementing checks at the circuit layer that are exposed to an end-user does not seem to be worth the time saved by not encoding them directly into the garbled circuit. This comes from our experience with our output consistency check, which was difficult to create correctly for each test program.
5. Ensure that all the features of a developed compiler and execution system are thoroughly unit tested.

4.6 Related Work

The creation of circuits for SFE in a fast and efficient manner is one of the central problems in the field. Previous compilers, from Fairplay [5] to KSS12, were based on the concept of creating a complete circuit and then optimizing it. PAL [6] improved such systems by using a simple template circuit, reducing memory usage by orders of magnitude. PCF [8] built from this and used a more advanced representation to reduce the disk space used. We use similar techniques to produce circuits with sizes comparable to that of Kreuter *et al.*. Our system also improves their runtime system to produce one that is significantly faster and uses much less bandwidth. Of course, none of these systems provides the ability to save state across computations.

Other methods for performing MPC involve homomorphic encryption [43, 44], secret sharing [45], and ordered binary decision diagrams [46]. A general privacy-preserving computation protocol that uses homomorphic encryption and was designed specifically for mobile devices can be found in [39]. There are also custom protocols designed for particular privacy-preserving computations; for example, Kamara et al. [47] showed how to scale server-aided Private Set Intersection to billion-element sets with a custom protocol.

Previous reusable garbled-circuit schemes include that of Brandão [52], which uses homomorphic encryption, Gentry *et al.* [51], which uses attribute-based functional encryption, and Goldwasser *et al.* [50], which introduces a succinct functional encryption scheme. These previous works are purely theoretical; none of them provides experimental performance analysis. There is also recent theoretical work on reusing encrypted garbled-circuit values [53, 54, 55] in the ORAM model; it uses a variety of techniques, including garbled circuits and identity-based encryption, to execute the underlying low-level operations (program state, read/write queries, etc.). Our scheme for reusing encrypted values is based on completely different techniques; it enables us to do new kinds of computations, thus expanding the set of things that can be

computed using garbled circuits.

The Quid-Pro-Quo-tocols system [57] allows fast execution with a single bit of leakage. The garbled circuit is executed twice, with the parties switching roles in the latter execution, then running a secure protocol to ensure that the output from both executions are equivalent; if this fails, a single bit may be leaked due to the selective failure attack.

Chapter 5

Frigate

5.1 Introduction

The creation of the Fairplay compiler [5] ignited the research community. In the time since, SFE compilers have improved performance by multiple orders of magnitude, reduced bandwidth overhead, and allowed for the generation and execution of circuits composed of tens of billions of gates [3, 6, 7, 8]. These efforts have brought SFE from the realm of mere theoretical interest to the verge of practicality; as an indicator of this fundamental change, DARPA is spending $60 million to support the transition of technologies such as SFE to practice [9].

Despite these improvements, current SFE compilers fail in two critical areas: they are unstable and incomplete compared to industry-standard compilers. Specifically, as we demonstrate, these compilers often break, and when they do work, can produce executables which generate incorrect results. It might also be possible for incorrect results to be verified as correct by the SFE protocol under such circumstances.

In this chapter, we present *Frigate*, an SFE compiler developed using design and testing methods from the compiler community. We name our compiler after the naval vessel, known for its speed, maneuverability, and adaptability for varying missions.

Our compiler is modular, extensible, and can support a variety of applications. It is also significantly faster than anything else currently available. We hope that the frigate's use as a convoy-escort ship will parallel the use of our compiler as a facilitator of future research into SFE systems. In this chapter, we:

- **Demonstrate systemic problems in the most popular SFE compilers:** We apply differential testing on five popular and available SFE compilers, and demonstrate a range of stability and output correctness problems in each of them.
- **Design and implement Frigate:** Our primary goal in creating Frigate is correctness, which we attempt to achieve through the use of principled and simple design, careful type checking and comprehensive validation testing. We use lessons learned from our study to develop principles for others to follow.
- **Design Frigate to be extensible:** Our secondary goal was to provide a compiler that can be extended to provide useful and innovative functionality. After we completed the compiler we added signed and unsigned types, typed constants, and three special operators.
- **Dramatically improve compiler and interpreter performance:** The result of our efforts is not simply correctness; rather, because of our simple design, we demonstrate markedly reduced compilation time (by as much as 447x compared with previous circuit compilers), interpretation time (by over 786x), and execution time (up to 21x) when compared to currently available systems. As such, our results demonstrate that principled design can create correct SFE compilers while still allowing high performance.

Although these SFE compilers may be considered by some to be "research code," they are being used extensively within the community and by others as the basis for developing secure applications and improved primitives. The corpus of compilers we test represent a large gamut, including the most recently published solutions. In all cases, we find issues with correctness or efficiency. The implications are considerable,

as the unreliable nature of many of these compilers makes testing new techniques extremely difficult. It is imperative that the community learns from these failures in design and implementation for the field to further advance. A reliable compiler is critical to this goal.

The remainder of this chapter is organized as follows: Section 5.2 provides a background in compilers for garbled circuits; Section 5.3 introduces techniques used to validate correctness; Section 5.4 describes our analysis of existing compilers; Section 5.5 defines principles for compiler design; Section 5.6 presents the design of Frigate; Section 5.7 presents our performance tests comparing Frigate to five widely-used SFE compilers; Section 5.8 discusses related work.

The material in this chapter appeared in preliminary form in [21].

5.2 Background

Since SFE was originally conceived, a variety of different techniques for solving SFE have been developed. Recent work has demonstrated that each technique can outperform the others in different setups (*e.g.*, number of participants, available network connection, type of function being evaluated) [23, 58, 59, 60]. In this chapter, we focus specifically on compilers for garbled circuits. Garbled-circuit protocols have been shown to perform optimally for two-party computation of functions that can be efficiently represented as Boolean circuits. While our experimental analysis examines the performance of the compiler in the context of garbled circuits, it is critical to note that this compiler can be used with *any* SFE technique that represents functions as Boolean circuits.

5.2.1 Circuit Compilers

Execution systems for garbled-circuit secure computation require functions represented as Boolean circuits. Due to this requirement, there have been several compilers created to generate circuit representations of common functions used to test this type of computation. These compilers take higher-level languages as input and transform them into a circuit representation. Writing the circuit files without using a compiler is tedious, inefficient, and will most likely result in incorrect circuits as they can have billions of gates.

First, it is possible for the generator to garble a circuit that does not evaluate the functionality agreed upon by both parties. To prevent this attack, the cut-&-choose construction requires that the generator garbled many copies of the circuit to be evaluated. Some fraction of these circuits are then "opened" and checked by the evaluator to ensure that they were garbled properly. The circuits that are not opened are then evaluated, and the output of the computation is set to the value output by a majority of the evaluated circuits.

Second, given multiple evaluation circuits from the cut-&-choose, the generator can provide inconsistent input values across the different circuits. Several techniques exist to ensure input consistency, including the claw-free construction [10] or using auxiliary circuits to hash and compare the generator's input values [4]. These checks will alert the evaluator and terminate the protocol if the generator is caught cheating.

Third, the generator may provide incorrect wire labels to the evaluator during the OT protocol. This attack, known as a selective failure attack, can cause the circuit to fail evaluation for certain input values, leaking information about the evaluator's input to the generator. To prevent this attack, a committing oblivious transfer can be used to allow the evaluator to check the correctness of her input wire labels. Other techniques for preventing this attack include encoding the evaluator's input in such a way that a selective failure is no longer tied to the evaluator's actual input value [4,32].

Finally, the evaluator may tamper with the output of the garbled circuit after she learns the real bit mappings from the generator. A number of output commitment protocols have been developed to prevent this attack, allowing the circuit to privately and securely output separate output values for each participant in the protocol.

5.3 Compiler Correctness

One of our main motivations for developing a principled compiler was the varying and unstable state of the existing research compiler space. Garbled circuit research has made significant advances in the past several years, which is largely due to a set of circuit compilers that have been commonly used to generate test applications for a significant number of protocols. Given our years of experience, we know the reliability of these results is suspect in many cases due to common errors we have found in these compilers. To facilitate continued advances in this research space, a foundational compiler with reliable performance is a critical tool. Without it, researchers will be forced to either use existing compilers, which we show are unreliable, or develop their own compilers, which is time-consuming and slows research progress. To demonstrate the need for a new and correct compiler that is openly available for the community, we examined correctness issues with the most popular and powerful compilers used in garbled circuit research.

We define the *correctness* of a complier implementation using two criteria: (1) any valid program in the language can be successfully compiled, and (2) the compiler creates the correct output program based on the input file. There are two methods used to demonstrate compiler correctness: formal methods for validation and verification, and validation by testing.

5.3.1 Formal Verification

The concept of a verifying compiler was identified as a grand challenge by Tony Hoare in 2003 [61] due to the significant complexity in design and implementation. Since that time, the primary example of a formally verified compiler has been CompCert [62]. The development and rigorous proof of each formalized component of the compiler was an immense undertaking. However, despite the amount of time and formal verification that went into CompCert, it was demonstrated that the formal verification used in CompCert was only able to ensure correctness in select components of the compiler. When tested with Csmith [63], there were still errors found that demonstrated the limitations of formal verification. In addition, formal verification of compiler transformations and optimizations is still very much an open research area [64, 65]. Techniques such as *translation validation* [66, 67, 68] focus on the formal validation of a compiler's correctness through the use of static analysis techniques to ensure that two programs have the same semantics, and are designed to attempt to deal with the reality of legacy compilers. They have their limitations as well, particularly within the context of secure multi-party computation compilers that have not adopted any particular standard for intermediate representations. As a result, the semantic model must be adapted for every compiler implementation, and any changes in the compiler require changes to the model.

Based on these limitations and the impracticality of applying formal verification, we instead apply validation techniques that are the standard method for ensuring the correctness of compilers.

5.3.2 Validation By Testing

Validation by testing attempts to demonstrate the correctness of a compiler through extensive unit testing. This, we note, is by far the most common technique used in practice to ensure compiler correctness. While testing for correctness can miss

some errors in compiling specific cases, it provides a practical level of assurance that is sufficient for the vast majority of applications. Validation tests are designed by examining how to rigorously test the largest possible number of programs a compiler can generate.

There are many existing validation tests [69, 70, 71] and test suites [72, 73]. The validation tests used by ARM [73] and SuperTest [72] provide a description of the procedures they use to validate the vast majority of possible program cases. However, these suites are language-specific, often developed to find errors in popular tools such as *gcc* and *LLVM*. To date, there have not been existing validation tools designed to examine secure computation compilers. As a result, we developed our own set of validation tests based on the techniques used by these tools. Our tests, like the test suites of ARM and SuperTest, explore the possible statements and effects of those statements.

In our case, hand written tests are preferred over automatically generated tests due to us being able to examine the compiler source directly. This allows us to examine possible code paths and be more systematic with our tests than a random fuzzer. In addition, because there are no SMC compiler standards, a different fuzz generator would have to be created for each compiler input language.

Our tests follow the concept of testing the state space of the compiler starting with broad examination of operators and expressions, then refining the tests to consider common special cases. Our tests proceed through five phases:

1. Attempt all possible grammar (syntax) rules and print out the results. This shows that the compiler reads in programs correctly and demonstrates the internal program state is correct.
2. Beginning from the simplest operation to validate correctness (*i.e.*, outputting a constant) test each operator in the language and each control structure to ensure it outputs the correct result.

(a) Test the different possible primitive types and declarations.

(b) Test each operator as to whether it creates the correct output circuit.

(c) Test each control structure by itself.

(d) Test function calls, parameters, and return statements. Verify that parameters can be used inside of their functions and that return statements work correctly. Also perform tests for where different types are used as input parameters and return values.

3. Validate all the different *paths* for how data can be input into operations. Demonstrate that different control structures work correctly together. Or, as put by SuperTest [72], "Systematically exploring combinations of operators, types, storage classes and constant values."

(a) Test if the operator deals correctly with the possible types of data that can be input as an operand.

(b) Test different types of control structures nested within each other.

(c) Test each operator under *if* conditionals with emphasis on operators that change variable values such as assignment (=), increment (++), and decrement (--).

4. Test edge cases in programs.

(a) Verify that empty functions do not crash on definition or call.

(b) Test array access and how arrays (and like operators) deal with edge cases, *i.e.*, out of bounds, minimum, and maximum values.

(c) Ensure known weaknesses in past compilers are tested to determine whether these vulnerabilities appear.

5. Perform testing to verify each previously found error was not re-added to the final implementation.

At the conclusion of these tests we have tested (1) the correctness of each mini-circuit an operator uses, (2) the ways data can come into each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, array declarations), (4)

various edge cases. This set of tests checks the operators, control flow structures, types, and covers most, if not almost all, of the uses of the operators, control flow structures, and types.

5.4 Survey of Existing Compilers

Using the test procedures described in the previous section, we set out to quantify the common problems in existing secure computation compilers. With each compiler, we found problems that would prevent the compiler from working correctly or corrupt test applications.

While the compilers we compare ourselves to are research artifacts, such systems are widely used by the community to test and validate algorithms. Having bugs and unpredictable behavior stunts the advancement of the field.

Upon acceptance, we informed the authors of each of these compilers about the issues we found; thus, many of the errors have been corrected in the interim.

While we do outperform many systems, the thrust of our compiler *not* mere performance. We compare ourselves to a number of well-known and widely tested systems and show that we are less prone to errors and produce good results. We ensured that our system was similar to current programming languages[1] and thus easy for new developers to use, and added descriptive error messages. All of this was designed to improve the way the community wrote SMC code and reduce complexity.

5.4.1 Comparison Compiler Information

Fairplay: Fairplay [5] was the first compiler to be used for practical research in secure computation. Fairplay's input format is SFDL, a custom hardware description, and the output is SHDL (simply a gate list in ASCII). We selected this compiler since it

1. We explain why we do not use a current programming language in Section 5.6.1.

initiated extensive practical research. We note some errors in the system, but we do not include it in our speed or other tests, since Fairplay is much older and slower, and running many of the test programs we use for the other systems is almost impossible because of memory usage; it would not be fair to compare it against current, state of the art compilers.

PAL: We selected PAL [6] as it was the first compiler designed for low-memory devices using an efficient intermediate representation. It also takes in Fairplay's [5] SFDL, a custom hardware description, and outputs SHDL, a gate list. It is dramatically more memory efficient than Fairplay and is able to compile much larger programs, but lacks optimizations used in recent work.

KSS: We examined the compiler from Kreuter *et al.* [3], hereon referred to as KSS. This compiler takes a hardware specific language as input and outputs a gate list in binary format. We chose KSS since it forms the basis for multiple recently-published works.

CBMC: The CBMC-GC compiler [7] (hereon CBMC) used a bounded model checker to compile a circuit program. This compiler takes a C file as input and outputs a condensed gate list (in ASCII). Because CBMC can compile programs written in ANSI C, it is commonly used in other garbled circuit research.

PCF: The PCF compiler [8], was created in order to have a condensed output format while being efficient. It takes in LCC bytecode as an input language and transforms it into a PCF file (ASCII). This file describes a circuit in a condensed format; a circuit interpreter is used to get each gate in turn. We selected PCF since it has been used to generated some of the largest circuits.

Obliv-C: Obliv-C [74] provides an extension that allows users to compile Obliv-C programs using a C compiler. These programs are then run in the normal C program fashion, *i.e.*, the default output is `a.out`. This has the advantage of being able to use the $-O3$ optimization flag.

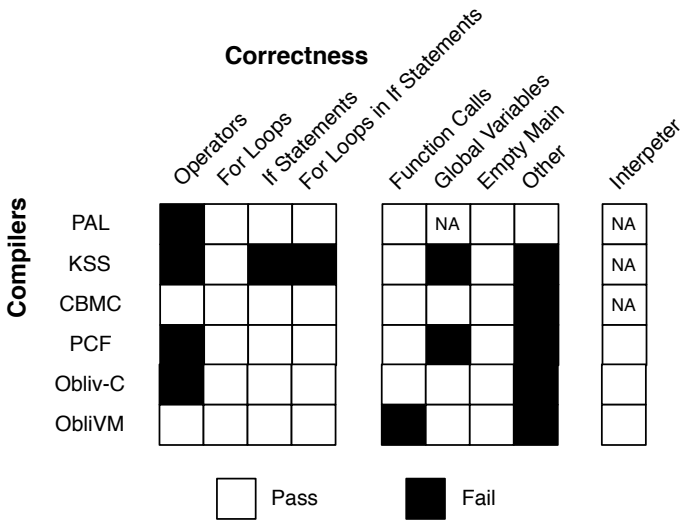


Figure 5.1: Summary of the correctness results.

ObliVM: ObliVM [75] compiles the input language into a Java file and then compiles the Java file using the `javac` Java compiler. The class file is then run using Java. By using Java, ObliVM is able to take advantage of the Java virtual machine optimizer.

5.4.2 Analyzing Compiler Correctness

We performed an analysis of the compilers to determine whether they work as expected, which we summarize in Figure 5.1. We do not attempt to find edge case errors that may only affect a minutiae of programs. Instead, we focus on testing the main operations and control structures in the input language that most users would perform.

We separate our analysis of previous compilers into two areas: errors and inefficiencies. We only note an error if the original program was valid; if the compiler crashes due to an incorrect program we do not consider it to be a compiler error. However, we found that most of the compilers lacked helpful error messages when an invalid program was provided as input. We communicated with the creators of these systems and provided details about the issues we encountered, as well as recommendations on how to correct them; many of the errors and ineffeciencies we outline

below have since been corrected or mitigated.

Fairplay

Note: We do not include Fairplay in our comparisons; we only list these issues here.
Errors: Fairplay sometimes outputs incorrect results when it encounters single depth *if* statements [6]. As *if* statements are needed in most programs, this may prevent most programs from working correctly. In addition, it frequently fails to correctly generate circuits for *for* loops with nested *if* statements. Common algorithms, such as Dijkstra's algorithm, require using *if* statements inside *for* loops.
Inefficiencies and Limitations: Fairplay's compilation process and circuit representation is the most inefficient we tested. The output circuit files are in ASCII, making them significantly larger than necessary. While this facilitates manual inspection of the circuit file, it made the storage requirements for these circuits far too costly for practical use. These inefficiencies and errors imply that Fairplay is only capable of compiling very simple programs that are too small to be relevant in the real world. For example, the AES circuit is a standard benchmark for modern secure computation systems. Fairplay terminates with an out-of-memory error when trying to compile AES even when given over 50GB RAM.

Fairplay's output does not use the most efficient adder sub-circuit circuit. This means it requires $3n$ AND gates instead of n. This affects all operations other than bitwise ones. It also uses a 3 input gate for MUXs. The newer and better MUX, uses 1 AND gates and 2 XOR gates for a reduction in truth table entries by 50% (for non-XOR gates).

PAL

Errors: PAL encounters problems when *struct*s are used. This prevents the use of complex data-types unless each data item is independently defined. This appears to

be an issue with the compiler's front-end, and is indicative of insufficient validation for that function within the language.

Inefficiencies and Limitations: PAL circuits are compiled into ASCII, which results in much larger file sizes than using a binary format. PAL also has some problem-size limitations, and fails to compile very large programs. While the templating concept proposed in this work is useful, it is not useful for compiling circuits of practical size.

PAL does not use the most efficient adder sub-circuit, meaning it requires $3n$ non-XOR gates instead of n (XOR gates can be executed for "free" [40]), and uses 3 input gates for MUXs. It also does not provide a complex optimization phase, so the output is not very optimized.

KSS

Errors: The KSS compiler has a number of areas where it does not function properly. Nested *if* statements consistently cause errors in the output circuits. Further, *for* loops used within *if* statements also cause the compiler to fail with regularity. Programs requiring multiple conditional statements, such as Dijkstra's algorithm, must be re-written to use single nested *if* statements. This is furthered hampered by the lack of conjunction operators. These shortcomings limit expressivity and the ability to write certain programs.

We discovered that when a variable is used inside of a function and also outside (*i.e.*, a global variable defined later in the code), it can lead to incorrect output circuits. This error can occur when a function and the body of the program might both use the same name for input. One such case occurred when we used the variable a both in the program body and in a function. Finally, we found a set of cases that the generated circuit was incorrect due to what may be an optimization error. This set of cases each used some of the same functionality.

Inefficiencies and Limitations: Rather than reduce the output size using an intermediate representation, KSS outputs the entire circuit. It also uses a very large amount of hardware-specific code, which makes porting a extremely difficult task. While this hardware-specific code provides some efficiency gain on specific platforms, it makes the task of extending the code very complex.

Like PAL, KSS's output does not take advantage of the most optimized adder sub-circuit. As a result, many circuits are about 3x larger (non-XOR gates) than necessary.

CBMC

Errors: Output variables can cause compilation errors if used more than once, or if read inside the program. This is a common occurrence when a function returns different values depending upon conditional statements, *e.g.*, a piecewise function. These errors can be avoided by careful programming. There is an error where input cannot be assigned directly to output variables. We also found another error that may be related to the input-to-output error though its exact cause is a mystery. CBMC provides an error message if an input variable is written within a program.

CBMC sometimes fails to compile using arrays as input. Without being able to rely on input arrays, the programmer must enter integers in an unstructured manner, making operations such as matrix multiplication more difficult.

Inefficiencies and Limitations: CBMC outputs the entire circuit in a ASCII format, which, while condensed compared to PAL, is still much larger than using binary. The format also doesn't map output variables to pins, making developing and debugging the interpreter prone to error.

PCF

This section tests the published version of PCF, PCF1. There is a new version under

development that, at the time of our testing, was not as efficient as PCF1.

Errors: PCF allows global variables, but if global values are initialized during declaration, it can crash, *i.e.*, the assignment must happen later on in the program. We discovered this error trying to get AES to work correctly, though this problem affects any program that uses global variables. When an array is addressed with an out of bounds index, PCF effectively returned random results (whatever was in memory) instead of producing an error message for each test we made. This is an extremely dangerous behavior, as it can lead to hidden and hard-to-detect errors.

By default, the PCF compiler does not update the program labels that keep track of the number input wires, meaning, by default, the amount of input will not be correct if these labels are used by an execution system. Furthermore, the programmer must calculate the input sizes (in bits) of each party in every program. The *translate* script provided with PCF, which is used to convert LCC bytecode to PCF, can fail on valid input files. In addition, PCF has input buffer overflow problems as inputs above 2^{14} bits overflow the input-buffers for the two parties. This means that the circuit will most likely fail upon execution when more than 2^{14} bits of input are requested in a program, like the millionaires problem with 65,536 bits as input. These input size bounds are currently hard-coded into the PCF compiler, not defined by the program being compiled, and must be edited manually in cases where larger inputs are needed.

Inefficiencies and Limitations: While PCF produces very small output circuits, the interpreter used to parse these circuits is extremely inefficient. Our tests demonstrated that the interpreter can require as many as ten operations to read in a single gate. This overhead is magnified by the fact that each gate is calculated by the interpreter for *every* circuit that is garbled. For malicious secure execution systems where many copies of the same circuit must be garbled, it is far more efficient to parse the gate once, then garble the same functionality as many times as are required for protocol security. PCF also produces spurious gates, which add to the circuit

```
obliv int a=0;
obliv if(input1>0)
{
        for(int i=0;i<3;i++)
                a = a + 1;
}
```

Figure 5.2: Example code from Obliv-C that does not optimize as much as it could.

complexity and should be removed. As with many other compilers studied, PCF uses ASCII output format, increasing storage size.

As PCF uses C as input, it also does not allow for arbitrary width types. In other words, it does not support, for instance, native multiplication of two 256-bit numbers.

Obliv-C

Errors: The statement `q = q & 0` throws a compiler error; multiplying a variable by 0 also causes an error. This type of multiplication is useful for eigenvectors. These operations work successfully if non-zero values are used.

Arrays going out of bounds (both access and modification) often produce no error messages or warnings, even those that should have been discovered at compile time. Hugely incorrect accesses (*e.g.*, out of bounds by a few hundred) can produce an error and crash, but smaller errors are often not detected. Such errors can affect the gate-count and modify the output in unexpected ways.

Further, the system cannot handle large arrays – the execution system crashed when we created an `unsigned int` array of size 32,000 for testing.

Inefficiencies and Limitations: Obliv-C does not always optimize circuits even when it is easily possible to do so. For example, `q = q & q` requires n gates, where n is the bitlength of the operation. In the segment of code seen in Figure 5.2, Obliv-C requires about 156 non-XOR gates; our compiler requires about 43 non-XOR gates for the same. It appears Obliv-C does not always keep track of optimizations for wire states at the gate level. These kinds of statements appear in programs such as the

edit distance of two strings. The authors suggest having the developer keep track of the possible range of an integer manually. However, this is not part of their compiler.

There are no arbitrary width types, which reduces the expressivity and increases gate counts of many programs. Trying to use a smaller type than an `int`, such as `char` produces seemingly strange gate counts, *i.e.,* the multiplication of two 8-bit chars and stored into an 8-bit `char` appears to not be an 8-bit multiplication and gives gate counts of either (1) a 32-bit multiplication or (2) a multiplication of unknown size (between 32 bits and 8 bits in length). We printed out the `CHAR_BIT` variable (the number of bits in a `char`) and used *sizeof* to verify the `char` is actually supposed to be 8 bits in length.

ObliVM

Errors: ObliVM provides a disclaimer on their code repository about the correctness of their system that it is expected to contain a variety of errors. In our attempt to get their code working, we encountered a problem with their test script to run their code. Through conversations with the manager of the code, this problem was resolved.

The typechecking is somewhat loose, *i.e.*, it allows many different lengths of variables to be used in an expression, *i.e.*, `int2 a; int4 b; int8 c; c = b + a`. The operator length appears to depend on the size of the output variable and not the size of the operands; this can lead to incorrect results if great care if not taken (*i.e.*, `int16 t = 4096; int8 q = t % 9;` results with q as 0 when 4096 mod 9 should be 1.). Safer type checking would eliminate this possible problem. We noticed this when we tried to write a modular exponentiation program. Single bit variables often throw errors when used; for instance, they cannot be combined with multi-bit variables (it throws a Java error).

When we tried to return (output) a result of size 2, but passed in a value larger than 2 bits, we received a result larger than would fit in 2 bits (*i.e.*, it appears the

```
int@n a=0;
if(input1$\char36$0$\char36$==1)
{
        a=a+1;
}
```

Figure 5.3: Example code: @n dictates the length of the variable, 0 picks the 0^{th} wire

```
error: incompatible types: t__T[] cannot be converted to int
->  int __tmp12 = f_tmp_6;}
```

Figure 5.4: Example error message from ObliVM.

return size may be ignored for the output).

The use of constants can sometimes be a problem. `x = 100+x;` throws an error, but `x = x+100;` compiles successfully.

When returning the result of an expression (*e.g.*, `x + y`), storing the value in a variable and then returning it (where the variable is of the return size) may produce a different value than returning the expression directly; both should produce the exact same result.

Inefficiencies and Limitations: ObliVM, like Obliv-C, does not appear to provide a large amount of gate-level optimization. The statements seen in Figure 5.3 require approximately $2n$ non-XOR gates (where n is the length of the variable). However, gate optimizations should prevent any non-XOR gates from being required in this segment of code. These kinds of statements appear in programs such as the edit distance of two strings. Likewise, a statement like `a = a & a` should require no gates of any kind, but it requires n AND gates in ObliVM.

Selecting more bits in a variable than exists allows compilation to succeed, but throws an error at runtime.

The error messages are not always helpful; they are mostly Java errors from the generated Java program. An example error can be seen in Figure 5.4.

5.4.3 Summary

PAL, KSS, CMBC, Obliv-C, ObliVM, and PCF crashed on programs that should correctly compile. KSS, ObliVM, and PCF generated incorrect circuits. These are important problems. Consider how easy it is for an array to go out of bounds or the number of programs that benefit from nested conditional statements. Or, if the expressivity is severely limited by incorrect operators then programs cannot be written as efficiently as they could otherwise. Principally, if the program files used in an SMC computation are not correct then the resulting SMC computation will not be correct either.

5.5 Compiler Development Principles

Given the problematic state of secure computation compilers in the research community, we set the primary goal of our work to be the development of structured design practices for secure computation compilers, and to demonstrate the effectiveness of these practices with a new compiler implementation. By examining practices used by the compiler community and combining those best practices with the observed failings of previous secure computation compilers, we have assembled a set of four principles to guide the development of our compiler, Frigate. Through this implementation, we demonstrate that these principles should be considered standard practice when developing new compilers for secure computation applications.

1. *Use standard compiler practices:* Use standard methodology from compilers (lexing, parsing, semantic analysis, and code generation). Use data structures that are described throughout compiler literature (*e.g.*, an abstract syntax tree) [76]. Applying these standard, well-studied constructs allows for straightforward modular treatment of the compiler components when extending the functionality. Furthermore, it allows for application of standard compiler debugging practices.

2. *Validate the compiler output:* All production compilers rely on proper program validation to ensure that the compiler functions correctly. A variety of validation test sets have been developed in both the research community and in industry that can be applied to newly-developed compilers [72, 73, 77].
3. *Handle errors well with helpful error messages:* Many sources describing good compiler practices emphasize the need to produce error messages, also known as negative results (e.g,. [72, 76]). While allowing the compiler to crash silently on an incorrect program does not affect its overall correctness, it severely hampers usefulness.
4. *Simplify the design:* A standard software engineering principle is to avoid erroneous code by using simple designs. This allows for more intuitive debugging when errors do occur, as well as facilitating the addition of future functionality.

5.6 The Frigate Compiler

To demonstrate the practical effectiveness of our compiler design principles, we designed the Frigate compiler and secure computation language. We also created a fast interpreter to read Frigate's output files efficiently. Our work demonstrates three additional contributions to the state of secure computation compiler research: (1) a new and simplified C-style language with specifically designed constructs and operators for producing efficient Boolean circuit representations; (2) a compiler that produces circuits with orders of magnitude less execution time than previous compilers; and (3) a novel circuit output format that provides an efficient balance between compact representation and speed of interpretation.

5.6.1 Input Language

Frigate's novel input language incorporates the best of what we have seen and used in the community and partially because of this, we can achieve substantial non-

XOR gate efficiency. Our novel output format provides a balance between file size and removing extra instructions necessary in some formats, *e.g.*, PCF. Frigate is meant to be a well-tested, user-friendly tool, which incorporates well known circuit optimizations and provides good performance, allowing researchers to easily create their own special optimizations without having to write their own compilers.

To better facilitate the development of programs that can be efficiently compiled into Boolean circuits, we developed a custom C-style language to represent secure computation programs. The language allows for efficiently defining arbitrary bit-length variables that translate readily into wire representation, and restricts operations in a manner that allows for full program functionality without excessive complexity. We do not use C or a common intermediate representation like LLVM's bytecode as input, to allow for innovative operators and non-standard bit-width operations.

This minimal set of operations adheres to our fourth design principle of maintaining simplicity to ensure for easier validation. Our language has control structures for functions, compound statements, *for* loops, and *if/else* statements. We include the ability to define types of arbitrary length and combination as in SFDL, the language used by Fairplay, combined with an operator that selects some bits from a variable used in the KSS compiler input language. We allow signed *int_t*, unsigned *uint_t*, and struct *struct_t* types in our input language (we can handle arrays inside structs). For modularity, we have *#include* statements to allow the use of external files and *#define* to replace a term with an expression. The list of operators in our language is in Table 5.1, with an example of our input language in Appendix B.1.

Every program begins with a declaration of the number of parties participating in the computation. Since not every participant is required to provide input or receive output, the input and output types for any subset of the participants may then be specified.

Operators	Description
+ - * / %	arithmetic operators
** // %%	extending and reducing operators
| ^ & ~	bitwise operators
=	assignment operator
++ −−	increment and decrement operators
== !=	equality test operators
> < <= >=	conditional operators
<< >>	shift operators
<<>	rotate left operator
.	struct operator
[]	array operator
{} {:}	wire operators

Table 5.1: A table showing the operators in Frigate's input language. As data types are either signed and unsigned, the arithmetic and conditional operations behave differently depending on whether the operands are signed or unsigned. In the case signed and unsigned types are used in the same operator, the compiler uses the unsigned operator (a warning is also issued by the compiler). Extending and reducing operators are discussed in Appendix B.2.

To further maintain simplicity, only three primitive types are defined in our programming language. *int_t* types are signed numbers defined to a specific bit length, *uint_t* types are unsigned numbers defined to a specific bit length, and *struct_t* types may consist of *uint_t* , *int_t*, and *struct_t* types. Developers may specify their own types using these three types and the *typedef* command. These three types can be combined to create any complex data type. To formally define the typing of each operator in our language, we give a selection of typing rules in Figure 5.5. The remainder of these rules are available in Appendix B.4.

One feature we were compelled to omit from our language was global variables. We removed this feature after we realized the significant overhead they represent within a Boolean circuit program. Allowing global variables requires keeping track of whether each function is called under an *if* statement and adding a MUX gate every time a global variable wire is assigned a value. Our language is capable of expressing equally functional programs by passing in "global" variables and returning any new values for these variables.

$$\textsc{Add}\ \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 + t_2 : Num_{L_i}} \qquad \textsc{Less}\ \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 < t_2 : Num_1} \qquad \textsc{Assn}\ \frac{\Gamma \vdash t_i : T}{\Gamma \vdash t_1 = t_2 : T}$$

$$\textsc{If-Else}\ \frac{\Gamma \vdash t_i : T \quad \sigma : Num_1}{\Gamma \vdash \texttt{if}\ (\sigma)\{t_1\}\ \texttt{else}\ \{t_2\} : T} \qquad \textsc{Func-Call}\ \frac{\Gamma \vdash t_i : T_i \quad f : F}{\Gamma \vdash f(t_0 ... t_{n-1}) : R}$$

Figure 5.5: Example typing rules for basic operators and control flow statements

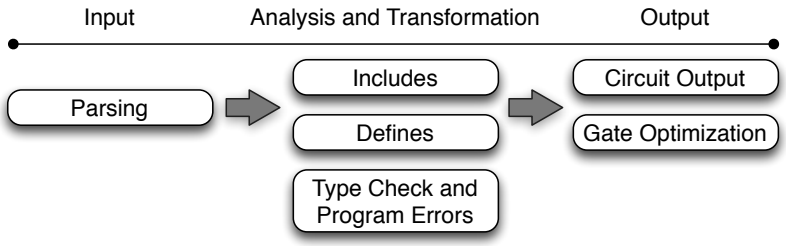


Figure 5.6: Overall design of the Frigate compiler. There are six separate blocks of the compiler separated blocks into three different stages instead of the traditional two stages.

5.6.2 Compiler Design

With our input language defined, we next examine the design of the Frigate compiler itself. Written in approximately 25,000 lines of C++, the compiler is designed to be simple enough to validate each output code path and modular for expansion to fit specialized secure computation applications. While there are other languages (with stronger typing, for instance), which would have made it easier to show the correctness of the compiler, we use C++ for speed and available libraries. Note that Frigate can handle a variety of security models since we can attach any SMC implementation to the compiler without affecting the adversarial model.

Compilation stages

Frigate represents programs in the standard compiler data structure, the *abstract syntax tree (AST)*. In accordance with our first design principle, this allows for straightforward static analysis and transformation of each program. Each type of operation has its own node where construction, type checking, and output of its sub-circuit

(among other functions) takes place.

Compilation of a program follows three phases as shown in Figure 5.6. The input section of Frigate takes in a program and creates an AST representation of the program. We used Flex [78] and Bison [79] to generate the scanner and parser used in this phase. In the second phase, any *#include* statements are replaced with the included file's generated AST. All *#define* statements replace any terms in the AST with a deep copy of the defined expression tree. To conclude this phase, the type checker takes the AST and checks that it is a valid program as defined by Frigate's input language. The final phase of compilation takes in the AST and outputs the circuit while performing gate-level optimizations. If a developer wishes to extend the functionality of Frigate, this modular phase design allows for additional stages to be inserted in between the existing stages.

Type Checking and Error Output

To satisfy our third design principle, we created our type checker to output detailed error messages to indicate the location and type of error generated by an incorrect program (*e.g., ./tests/add.wir, Error line:11 Type "mytype" is used but not defined*). To ensure developers do not include unstable functionality in their programs, Frigate enforces strict type checking that prevents different types from interacting unless those types are different signed or unsigned integer types of the same length. A warning is issued in this case.

5.6.3 Circuit Representation

Previous work in compiler development has demonstrated that it is possible to have either a large yet simple circuit representation that is efficient to parse, or a highly compact circuit representation that incurs significant cost when interpreted by the evaluator. To strike a balance between these two extremes, we developed a novel

circuit representation that is significantly smaller than the simplified circuit representations while still being efficiently parseable. Our output format represents circuits using four elements: a set of input and output calls, gate instructions, function calls, and copy instructions. Our representation of functions, as different files, allows us to shrink the output size without the need for a costly circuit interpreter (more details in Section 5.6.3).

To further improve the efficiency of evaluating Frigate circuits, we designed the compiler to favor XOR gates, as they can be evaluated with fewer operations and do not consume bandwidth when certain garbled circuit protocol optimizations are used [40]. We use the four-XOR, one AND-full adder introduced by Boyar et al. [80].

Output Components

Here we present the details of our circuit representation.

Wires: Each variable is composed of many wires that are allocated as needed with a set address. Each wire exists in either a *used* wire bin or a *free* wire bin. Once a *used* wire is freed it is placed in the *free* bin. Order, as defined by the address of a wire, is not preserved in the *free* wire bin. Our compiler will free the wires it can after each operation.

We group wires together by the number requested for a specific variable. This allows for a massive decrease in the amount of time required for checking whether or not wires can be placed in the free wire bin, *i.e.*, instead of requiring 100,000 checks for a variable with 100,000 bits (wires) in length, only a single check is needed.

Wires can exist in one of six states. *ZERO* and *ONE* represent a wire's state as 0 or 1. The *UNKNOWN* state represents wires that depend on input values such that their value cannot be computed at compile time. *UNKNOWN_INVERT* represents an unknown wire but at some point was inverted. *UNKNOWN_OTHER* and *UNKNOWN_INVERT_OTHER* are wires whose values are pointers to another

wire value or the inversion of another wire value. By keeping track of inverted states, we can optimize away inverts in some cases.

Gate Output: Given two input wires and a truth table, the *outputGate* function will output a gate and update the state of the output wire. An additional function is called to determine whether the gate is needed or whether it can be optimized out. If the gate cannot be optimized out then the truth table will be adjusted for whether either of the input wires' states are inverted. Finally, the gate will be added to the output.

Function Parameters and Return States: Since we output the gate representation of each function independently only once, uncorrelated with a single function call, we cannot take advantage of knowing the state of a wire as it is passed into a function. Therefore, function parameter states are marked as *UNKNOWN*. It is possible to pass wires with "0" and "1" states, but it is not as efficient as the optimizer cannot use the information that they are "0" and "1" since they must be marked as *UNKNOWN*. This inefficiency is necessary since we only output each function a single time preventing us from taking advantage of specific parameter states. We could solve this by outputting multiple function files with different wire parameters, but this would expand our circuit representation.

Circuit Interpreter

Using our circuit output format, the process of interpreting a circuit is reduced to a highly efficient task. When the interpreter is initially called, it reads an *.mfrig* file, which contains information about the number of parties, input and output sizes, and the number of functions. It then opens the *.ffrig* function files. After the interpreter is initialized, it is ready for the first *getNextGate* command. Each time *getNextGate* is called, the compiler reads and executes the next instruction, and returns the appropriate gate to the execution environment.

Each function occupies a specific set of wire values such that no function's wires will overlap. This enables a "stack" of function calls without the need for the push and pop operations that would be required if our functions used overlapped wire addresses. This does not affect the output circuit size. The interpreter keeps a call stack of active functions in its internal state. Each function, rather than being held completely in memory, is stored as a pointer to the active instruction. When a function is called, the stack of functions is updated, the active function is set to the called function, and the called function is set back to the first instruction.

5.6.4 Procedures

While our technique of dividing programs into distinct functions and then composing the circuit with calls to those functions allows for a significant reduction in the representation size of many circuits, not all programs can be easily partitioned into distinct functions. Even if a clean partitioning does exist, the function overhead for copying parameters and return values can exceed the number of commands inside the function. Large representation size is commonly encountered with loops, creating redundant data that expands the size of the circuit representation. To reduce the output file size in this case, we develop a novel construct called *procedures*.

A procedure is an area of a loop that can be moved by the compiler to a separate function so that, instead of unrolling all the instructions for every iteration of the loop, all that is required is a single function call to the procedure function. Procedure can be intermixed with other non-procedures inside of the same loop.

As the procedure circuit is exactly the same each iteration, there are limits when using variables whose values are *ONE* or *ZERO* inside of the procedure and change between iterations. Most notably, this limitation includes using the value of the loop variable.

To demonstrate the output file size reduction possible using procedures, we con-

sider an example program that adds five 32-bit variables to an accumulator 1000 times (the full program is in Appendix B.1). If no procedure is used, this program requires an output file of about 13MB since each iteration of the main loop must be unrolled. However, if a procedure is used, then output is one 30 KB file (main) and one 13KB function file (the procedure), a reduction of the total disk usage by over 300x.

5.7 Experiments

5.7.1 Frigate Validation

During the creation of Frigate, we unit tested each new structure to ensure it functioned properly. Our unit tests comprised checking most, if not all, possible program paths. We manually checked each operator with sample output. Then, to demonstrate the correctness of circuits created by Frigate, we ran an extensive validation test suite consisting of over 17,000 tests and several million additional tests containing all possible combinations of input using 8-bit types for complex operators. After hundreds of iterations of development and testing and months of work, Frigate successfully passed all validation tests, and produces correct and functioning circuits in every case where previous compilers failed. For further details on the state space we examined in Frigate, see Appendix B.3.

5.7.2 Compiler Efficiency Tests

By constructing a compiler using our four development principles, we wanted to evaluate whether adhering to the principles we laid out would have an adverse effect on performance. We tested the time that is required to compile circuits in Frigate against the three compilers (CBMC, PCF, and KSS) that output a complete circuit. We also tested ObliVM and Obliv-C, but do not include them in the compile-time results as

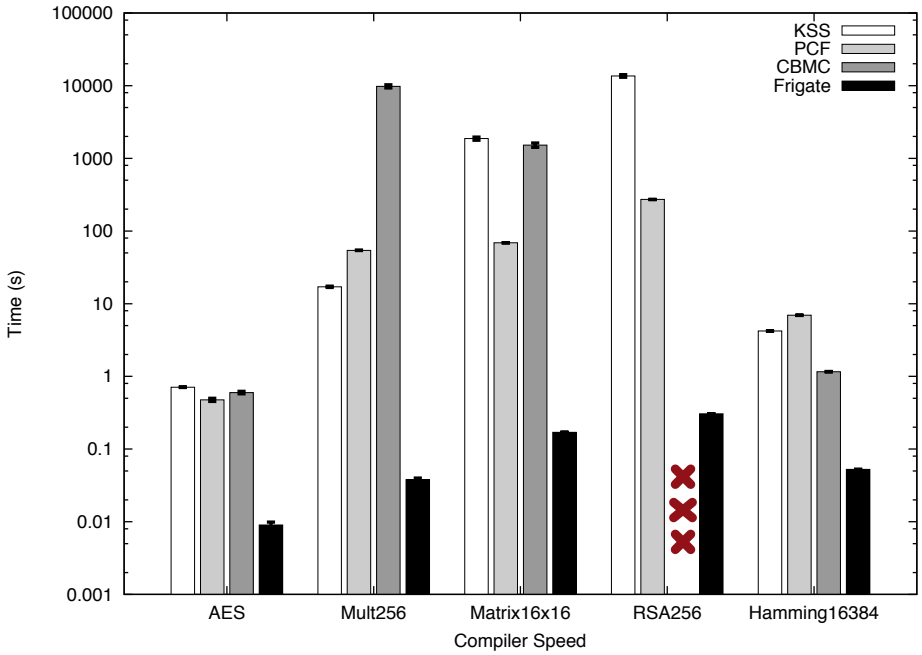


Figure 5.7: Comparing the different compilers we tested for compilation time. We did not succeed in compiling RSA256 with CBMC. Note the y-axis is logscale.

they do not directly output circuits and thus are not directly comparable. We show some of these results in Appendix B.1. PAL did not give competitive compilation results, so we omit them from our benchmarks. For Obliv-C and ObliVM, we measure the efficiency (gate counts) of primitive operations as the runtimes of the compilers correspond to the C and Java compilers, respectively. All of our benchmarking tests were performed on a MacBook Pro with an Intel i7 4-core 2.3Ghz with 16GB RAM, 256KB L2/core, and 6MB L3.

Test Programs

To evaluate performance across a wide variety of compilers, we used common test programs used by the other researchers in this space [3, 4, 8]. We used the following test programs: multiplication with matrices of X by X with 32-bit values, AES, Hamming distance of two X bit numbers, multiplication of two X-bit numbers that produces a 2*X-bit result (this is in contrast to the program of TinyGarble [81] who

use X*X=X-bit multiplication), and RSA (modular exponentiation) of X bits, where the base, exponent, and modulus are all X bits in length. For each test program, we varied the input size X throughout our testing.

Tests

We summarize the results in Figure 5.7 by comparing the largest input values for each program that successfully compiled across all compilers. We evaluate the compilers with their default setup in an attempt to produce the smallest circuit (as opposed to disabling the circuit optimizers). In every case, Frigate completes compilation the fastest. In the best case, Mult 256, which computes the multiplication of two 256-bit numbers, Frigate compiles 447x faster than the next fastest compiler, KSS.

In addition to comparing speed efficiency, we also considered the non-XOR gate counts of each program compiled. Because the free-XOR optimization for garbled circuits [40] allows XOR gates to be evaluated with non-cryptographic operations and without consuming network bandwidth, we consider non-XOR gates the bottleneck in computation. Frigate greatly reduces the number of non-XOR gates for the Mult-4096 program, demonstrating a reduction for the number of non-XOR gates by about 3x. In the case of AES, and RSA-512, the improvement was only slightly better than existing compilers, reducing the gate count by up to 1.18x. We observed a increase in gate counts for Matrix Multiplication by 0.8%, Hamming Distance by 2.35x, RSA-256 by 1.19x, and Mult-256 by 1.37x. The full compilation results are in Table B.1, in the Appendix.

Other than Hamming Distance and AES, our gate counts are similar to the best gate counts of [81], who wrote programs in behavioral and RTL level Verilog. For the three test programs given in [81] that use a high level language (C), we are superior. Table 5.2 gives the exact results.

While TinyGarble produces superior gatecounts in some cases, this is achieved

	Frigate	TinyGarble	
ProgramName		C	Verlog
Hamming-160	719	1,264	158
Sum-1024	1,025	3,067	1,023
Compare-16384	16,386	52,224	16,384
X-to-X-bit Mult-64	4,035	-	3,925
MatrixMult5x5	128,252	-	120,125
AES	10,383	-	5,760

Table 5.2: Non-XOR gate count comparison between Frigate and TinyGarble [81] using HDL and C as inputs. "-" represents results not present in [81]. For accurate comparison, our multiplication operation in this test produces n-bit output as in [81].

using Verilog, a hardware description language for electronic systems. Thus, the interpreter converts something that is already close to a hardware-level description into a circuit format, as opposed to dealing with a high level language. It is not surprising, then, that in a some cases, the Verilog version, which is closer to a handcrafted circuit, performs better than Frigate.

For Obliv-C and ObliVM, we measure the cost of some primitive operations, shown in Table 5.3. Frigate, Obliv-C, and ObliVM have similar gatecounts for compare and sum operations. These operations have $O(N)$ gates, where N is the bitlength of the operation. In contrast, ObliVM's multiplication and division templates are larger than that of both Frigate and Obliv-C, and Frigate's division template is larger than that of Obliv-C.

It should be noted that neither Obliv-C nor ObliVM were able to perform the a = a & a optimization to emit AND gates. We include this optimization to show that neither of these two systems perform a number of known optimizations that should have been included. This is a disadvantage of their model of compiling to an executable instead of a circuit: in order to perform these optimizations, the system will have to perform them every time the circuit is executed.

To summarize, *ObliVM and Obliv-C do not perform known gate-level optimizations*. Without these and many other optimizations implemented in Frigate, they sometimes produce comparatively inefficient output programs.

	Frigate	Obliv-C	ObliVM
Sum-32	31	31	32
Compare-32 (>)	32	32	32
X-to-X-bit Mult-8	59	-	120
X-to-2X-bit Mult-8	136	-	176
X-to-X-bit Mult-32	995	993	2,016
X-to-2X-bit Mult-32	2,082	-	3,008
Div-8	61	-	172
Div-32	1,437	1,210	2,236
a = a & a	0	$O(N)$	$O(N)$

Table 5.3: Non-XOR gate count comparison of different operations for Frigate, Obliv-C, and ObliVM. For these tests we look at the non-XOR gate counts different primitive operations require (not gate counts for a specific program). For Obliv-C, we do not measure 8-bit operations (`char` variables) as they does not appear to give correct gatecounts as noted in Section 5.4.2. Using signed types.

5.7.3 Interpreter and Execution Speed

Interpreter Time: Our next set of experiments compares the performance of the Frigate and PCF interpreters. Figure 5.8 shows our experimental results. The Frigate output format allows for significant reduction in interpreting time. In the worst-case, we improve over PCF by 106x, with a reduction of 786x in the best case. We used the Unix *time* function to measure the total computation time.

A lot of this speedup comes from the way Frigate and PCF are designed. (1) Frigate optimizes the circuit a single time in the compiler; PCF has to optimize at runtime. (2) Frigate's execution system can take advantage of certain compiler (*gcc*) optimizations that PCF cannot due to a design decision requiring each instruction to use function pointers (*i.e.*, lots of function calls that could be optimized out). (3) Many instructions in PCF require a malloc due to the interface to the PCF interpreter; Frigate's interpreter requires no mallocs after initialization.

Execution: The total execution time is improved by a faster interpreter. We connected the Frigate interpreter and the PCF interpreter into the same semi-honest execution system loosely based on the KSS execution implementation but further optimized and modified to use generic C++ vectors as its primitive type (instead of

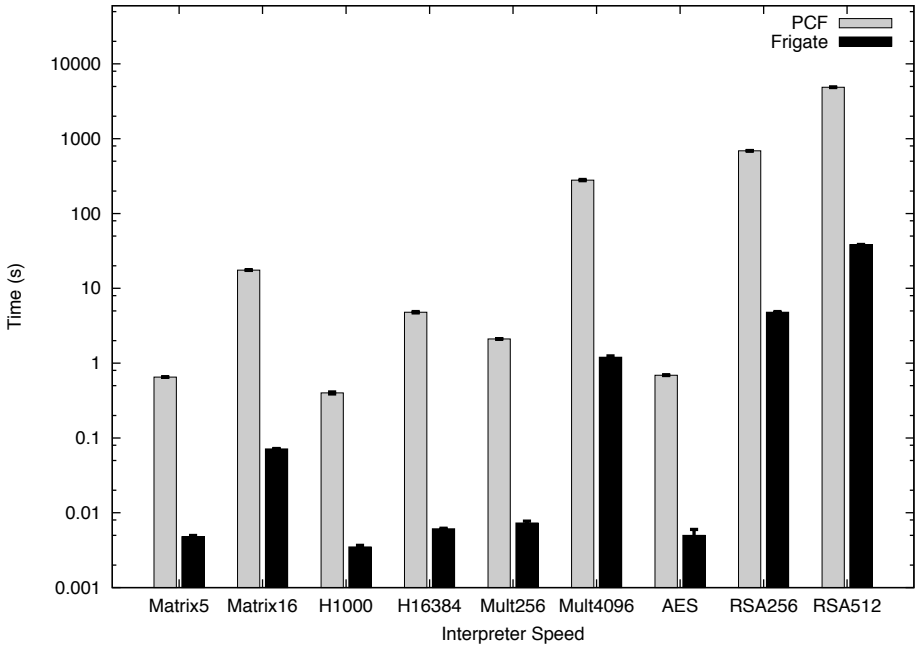


Figure 5.8: Interpreter time per circuit for PCF and Frigate interpreters. Note the y-axis is logscale. H stands for Hamming Distance.

Intel-only intrinsic data types) to increase portability. Figure 5.9 shows the results.

Total execution time improves by 21x in the best case and by 1.8x at worst. We can further improve our performance by reducing the overall execution time by adding in additional optimizations like the half-gate optimization [82] or the fixed-key blockcipher optimization [83]. The speedup is the result of the amount of extraneous instructions PCF requires, sometimes up to 18x instructions per gate instruction. The interface, though elegant, requires malloc on many of these. Here, our novel output format provides an advantage.

5.7.4 Discussion

Speed of Frigate: During the creation of Frigate, we attempted to speed up Frigate in many ways. Other than writing efficient code, our separation of functions, output representation, choice of programming language, efficient data structures, lack of a global optimization phase, efficient use of circuit templates, and our use of *procedures*

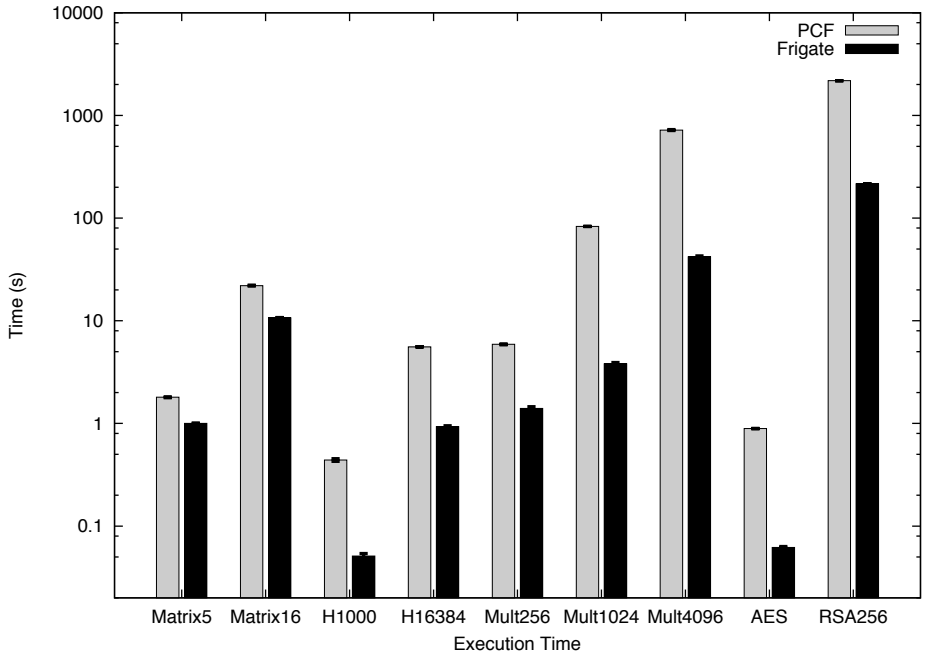


Figure 5.9: Execution performance for semi-honest execution system in Frigate and PCF. In these experiments we only vary the interpreter and circuit format. The execution system is the same in both cases. H stands for Hamming Distance.

all contribute to why our compiler performed better.

Not Comparing to Other Interpreters: Both Obliv-C and ObliVM compile to executables and not directly to circuits. For PCF and Frigate, we could easily swap the interpreter while maintaining the same SMC execution backend.

Extensibility: After the initial creation and implementation of Frigate, we made additional changes that show extensibility. We enabled constants to be defined with a specific sign and specific bit-length, which was not in our original specification, and added three additional operators: extending multiplication, reducing division, and reducing modulus. These operators are discussed in detail in Appendix B.2.

For developers to extend Frigate with their own functionality, they simply create or modify an AST node and the parsing rules, modify typing for new or existing operators, and then define what sub-circuit the operator outputs.

Tools: To demonstrate how Frigate can be used to create useful developer tools, we created an extension to output the gate counts of program components inline in a

```
function int mul(int x, int y)
<1047555,2092036>{
  return x * y;
}
function void main()
<270270213,540799236>{
  int t = input1;
  for(int i = 0; i < 256; i++)
  <268174080,536610048>{
    t = mul(t, input1);
  }
  <1047555,2092036>{
    t = t * input1;
  }
  output1 = input1 * input2 + t;
}
```

Figure 5.10: Example of Frigate's gate counts in the program at each compound statement. Key: $\langle non{-}XOR\ gates, free\ operations \rangle$

printout. We implemented this tool to understand where the most costly operations are in a program. Our tool also maps in the cost of function calls even though they are not called during compilation. As procedures are not output during each iteration of a loop, the costs inside a procedure represent only a single iteration, but outside the procedure, it is counted for every iteration. Figure 5.10 shows an example.

Very Large Circuits: Using our unique output format, we were able to compile a program that has 2^{467} non-XOR gates in under 20 minutes (a 1024-bit addition performed 20000^{32} times). This is not the limit, but shows Frigate can create very large programs.

Full Circuit List: Since many systems we have seen and used require the full circuit as input, we created a tool (via a command-line argument to the interpreter) to output the complete circuit into an easy to understand text format.

5.8 Related Work

When the garbled circuit protocol was developed by Yao [1], it demonstrated that secure multiparty computation was possible. However, the protocol remained a theoretical novelty until Fairplay [5] demonstrated that the protocol could be feasibly run for small circuits. In the time since, there have been many other compilers built for

SMPC [7, 11, 12, 13, 14], with steadily improving performance. Our work draws from many of these to produce a compiler that is significantly faster and more stable than any other system currently available.

Several optimizations have been developed to reduce the size of garbled circuits. The free-XOR technique [40, 84] allows garbled XOR gates to be evaluated with a single XOR operation and requires zero bandwidth. Optimizations such as garbled row-reduction [41] allow for the size of the transmitted AND gates to be reduced by a constant factor. Other optimizations, such as FleXOR [85], have been shown to reduce bandwidth and computation time for certain functions. The pipelining technique developed by Huang et al. [2] generates and transmits the circuit in layers, allowing large circuits to be handled in a small amount of memory. Most recently, the PartialGC system [20] allows for garbled wire values to be re-used between protocol executions. However, while these protocol optimizations allow for constant factor improvements in speed and bandwidth, they do not optimize the size of the boolean representation itself, which we do in this work.

Fairplay [5] was the first SMC compiler. While this provided a first step towards a practical and usable means for representing arbitrary programs as circuits, it suffered from a number of correctness issues. To reduce the size of the unoptimized circuit representation, the PAL compiler [6] used pre-optimized templates instead of completely creating each circuit at runtime. The compiler by Kreuter, shelat, and Shen [3] incorporated some of circuit optimizations. The Portable Circuit Format compiler (PCF) [8] combined the concept of templating with several circuit optimizations. Another compiler, Wysteria [86], provides support for mixed-mode secure computation. Frigate combines many such optimizations and is designed to be extensible, so that future optimizations can be included with ease.

A recent work by Songhori et. al [81] shows how to use hardware tools to create SMC circuits. These produce significantly smaller output files and often significantly

smaller non-XOR gate counts when writing in an HDL language.

Chapter 6

Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation

6.1 Introduction

While significant advances in both the performance of and the security provided by SMPC protocols and their underlying primitives has improved over the past decade [7, 8, 10, 11, 12, 13, 14], the costs remain too high for many practical applications.

An emerging hardware primitive may help to dramatically reduce the cost of such computation. Intel's Software Guard Extensions (SGX) [87, 88] is an extension to the Intel architecture that enables the creation of secure containers called *enclaves*. An enclave is essentially a "reverse sandbox" enforced by hardware. Recall that a sandbox is an isolated execution environment in which none of the code running inside the sandbox can modify anything outside of the sandbox. The SGX "reverse sandbox" has the complementary property that code running outside of it cannot manipulate anything within it. Specifically, the code, data, and stack within the

enclave are protected. SGX can create a *measurement*, which is a cryptographic hash of the code and data within an enclave, ensuring that any unauthorized changes can be detected. Data that are corrupted or modified outside of the SGX enclave cannot affect anything inside the enclave, with the one exception that a process running inside the enclave may be forced to abort if a data item that it attempts to retrieve from outside the enclave is unavailable.

Furthermore, an SGX system (*i.e.*, a computer equipped with an SGX module) can use the measurement and other data to produce an *attestation*, allowing an external entity to verify that an enclave does what it claims and has been properly instantiated on a valid Intel platform. Secure channels can be created, allowing data to be shared securely between remote enclaves without revealing the contents to the untrusted code running outside the receiving enclave. If data need to be stored outside an enclave for later use, SGX can use *sealing* to securely encrypt the data for storage using a persistent hardware-based key. The data can be decrypted once the enclave (or, depending upon the type of sealing, another enclave) is deployed.

The introduction of SGX, which provide an environment for the isolated execution of code and handling of data, offers an opportunity to overcome such performance concerns. In this chapter, we explore the challenges of achieving security guarantees similar to those found in traditional 2P-SFE systems. After demonstrating a number of critical concerns, we develop two protocols for secure computation in the semi-honest model on this platform: one in which both parties are SGX-enabled and a second in which only one party has direct access to this hardware. We then show how these protocols can be made secure in the malicious model.

Our goal is to determine whether, and if so how, the capabilities provided by SGX can make 2P-SFE more efficient and/or more secure. These are nontrivial questions, because naïve attempts to use SGX to implement 2P-SFE can lead to information leakage. This is not a shortcoming of SGX, which is a general platform

meant to enable a wide range of computations. Rather, it is our motivation for careful analysis of the problems one encounters when using SGX to implement 2P-SFE and possible solutions. We conclude that implementing 2P-SFE on SGX-enabled devices can render it more practical for a wide range of applications. This analysis is an important step in understanding the capabilities and limitations of SGX (and SGX-like) modules.

Our contributions:

- We carefully examine the difficulties inherent in using SGX for 2P-SFE. For example, we show that timing and memory-access side channels can cause information leakage when an SGX system is used to evaluate a 2P-SFE protocol.
- We show how to augment an SGX system to provide stronger guarantees, and we provide a protocol that enables two SGX systems to perform 2P-SFE efficiently. We also provide a protocol for securely outsourcing the 2P-SFE SGX computation from a low-end device that does not have an SGX module to another device that does. This allows us to take advantage of a remote SGX hardware unit without requiring the presence of a local one.
- We show how to modify 2P-SFE protocols secure against honest-but-curious adversaries so that, when run on augmented SGX machines, they are secure against malicious adversaries.
- We describe a number of novel use cases for SGX, both with and without our augmentations.

The remainder of this chapter is organized as follows: Section 6.2 provides technical background on 2P-SFE and SGX. Section 6.3 explains some problems that arise in straightforward attempts to use SGX to implement 2P-SFE. Section 6.4 describes how to augment SGX so that it can be used to implement 2P-SFE, provides a secure outsourcing protocol for non-SGX machines, and shows how 2P-SFE and SGX can be efficiently used in conjunction to provide better security. Section 6.5 provides

performance analysis and experimental results. Finally, Section 6.6 discusses related work.

The material in this chapter appeared in preliminary form in [22].

6.2 Technical Background

The Software Guard extensions (SGX) module allows parts of programs to be executed inside of logically separated segments of the CPU called *enclaves*. An enclave is a general-purpose module (unlike, say, a DRM module) and can be used for any kind of program. SGX provides a hardware-based guarantee that the programs and memory inside an enclave cannot be read or modified by any programs outside of the enclave (including a program in different enclave). In particular, neither `root` nor any other type of special-access program can read or modify the memory inside an enclave.

Technically, the data inside of an enclave are still within the same registers and cache as other programs; however, SGX processors provide functionality to prevent unauthorized access.

It should be impossible for an adversary to determine what is accessed inside of the enclave or what is written back to RAM when the cache is full. Therefore, any data in the enclave that must be written back to main memory is encrypted and signed to ensure it cannot be read or modified by another program. Modifications to code, data, or stack outside an enclave cannot interfere with the operation of the enclave except in one way: if something needed by a program in the enclave is simply unavailable or has been corrupted, then the program may have to abort.

Comprehensive overviews of SGX can be found in Intel's whitepapers [87, 88]. Design of systems and protocols that make extensive use of SGX is covered by, *e.g.*, Baumann *et al.* [89] and Schuster *et al.* [90].

6.2.1 Towards Using Secure Hardware for Garbled-Circuit Protocols

Both garbled circuits and SGX are designed to be used in scenarios in which parties have private data that they want to give as input to a computation in such a manner that they receive the result of the computation, while no one else learns either the input or the result. Therefore, it is natural to consider using SGX-enabled machines to execute a garbled-circuit protocol. The reason that it is not straightforward to do so is that garbled circuits and SGX use different techniques to protect private inputs.

In garbled-circuit protocols (and SFE more generally), cryptographic guarantees are used to ensure the privacy of the data. In SGX, users rely on secure hardware to guarantee data privacy. SGX provides security against malicious adversaries as long as one trusts Intel's setup process. In the SFE world, this is comparable to having a trusted setup, on top of which one runs one's protocol (here, part of the "setup" occurs at the Intel factory when the hardware and private key are created). The security properties of the exact model used by SGX are described in Intel's whitepapers [87, 88].

6.3 Why Simple "Solutions" Do Not Quite Work

The security guarantees provided by SGX do not immediately translate into being able to perform 2P-SFE protocols in general or even garbled-circuit protocols in particular. Simple solutions that use unmodified SGX primitives may leak information or, in some cases, undermine the security of other code running under SGX. In this section, we explain how that can happen.

6.3.1 A simple 2P-SFE protocol implemented with SGX

Below, we describe a naive, straw-man protocol for performing SGX-supported 2P-SFE. There exist numerous ways of doing this, but almost all of them suffer from a number of problems that we discuss in the next subsection.

Setup: We start with the standard 2P-SFE setup – two mutually distrustful parties with private inputs who wish to jointly compute a function and produce private results. In this scenario, both parties have SGX-enabled machines and have agreed to run a specific program. The two parties are as follows: the *evaluator*, who will use his SGX module to evaluate the program, and the *sender*, who will check the agreed upon program and then send her input. In the following, a superscripted "+" denotes a public key, while a superscripted "−" denotes a private key that does not leave the SGX enclave.

Protocol

1. The sender ensures the evaluator will evaluate the correct program, $prog_{sgx}$, by checking the signed measurement, $Ecv^{eval}_{measure}$, from the evaluator's enclave. $Ecv^{eval}_{measure}$ is signed by the evaluator enclave's private key $Ecv^{-}_{key_{eval}}$.
2. The sender encrypts her input, $input_{sender}$, under the evaluator enclave's public key, $Ecv^{+}_{key_{eval}}$, and sends it to the evaluator.
3. The sender's encrypted input, $Enc(input_{sender})$ is decrypted inside of the evaluator's enclave using $Ecv^{-}_{key_{eval}}$.
4. The evaluator enters his own input, $input_{eval}$ into the enclave.
5. The enclave puts $input_{sender}$ and $input_{eval}$ into the SGX program, $prog_{sgx}$. It then executes $prog_{sgx}$ and encrypts the sender's output, $output_{sender}$, under the sender enclave's public key, $Ecv^{+}_{key_{send}}$.
6. The evaluator's enclave releases the evaluator's output to him, and sends the sender's encrypted output, $Enc(output_{sender})$, to the sender.
7. The sender decrypts $Enc(output_{sender})$ using $Ecv^{-}_{key_{send}}$.

6.3.2 Problems with simple SGX-supported 2P-SFE

Side channels

1. **Runtime:** 2P-SFE protocols are not vulnerable to timing attacks. This is achieved by ensuring all program paths take equal time, at the cost of efficiency. In SGX-supported 2P-SFE, if a secret value x determines the number of times, for instance, a loop is executed, the timing could easily narrow the range of x. Principally, an attacker could execute the same program offline with many different iterations of the same loop inside of the enclave to see how long several different numbers of iterations take. This may provide a lot of information if each iteration of the loop is easily identifiable, *e.g.*, if each iteration takes a second to execute.
2. **RAM Access:** Data access is not hidden in SGX-supported 2P-SFE, which can potentially leak significant amounts of data. *E.g.*, a simple database style query using a binary search, where one side, the client, sends a private query to check whether a given value exists within the database. The enclave on the server reads in the plaintext records and matches them, one by one, to the queried value. In such a scenario, the data access alone is enough to reveal a lot of information on the queried value. (If the query matched, we have the value itself, if not, we know that it lies within a certain range.) There exist some methods to add hardware-level cryptographic support to FPGAs [91], but not for RAM. The best ways to make RAM secure are still Oblivious RAM and similar techniques [92].
3. **RAM Timing:** A timing attack could reveal a lot of information about the item being queried in the binary search. If the item is located on the first jump, we know that it's the value in the middle, etc.

Cryptography vs Memory Out of Bounds

Garbled circuits rely upon cryptography for data privacy – information leakage is not an issue as we have proofs for correctness and security. While it is theoretically possible to "leak" data by simply outputting it in the predefined program, such a blatant problem is easy to notice. SGX, if used improperly, might leak information if memory goes out of bounds, which is one of the most common bugs in everyday programming [90]. This is among the most frequent errors in programming, and can have catastrophic consequences [93, 94]. Unfortunately, in SGX, such an error would not only break the security of the program (and enclave) in question, but would also affect the security of SGX as a whole, since users might be able to access or modify data that they should not be able to see.

Trusting SGX vs Trusting Cryptography

SGX requires the users to trust that the evaluator of the program has not broken into the enclave to watch the memory and that the supply chain was not disrupted with insecure parts. These might be acceptable assumptions for users that buy commodity machines, but not for nation states or large companies that could benefit from stealing secrets. In contrast, 2P-SFE protocols provide cryptographic guarantees.

2P-SFE protocols prove themselves equivalent to the "ideal model," which uses a trusted third party who will correctly obey the protocol. SGX uses the trusted platform model, which is weaker than the trusted third party model, since it allows side-channel and information flow attacks.

SGX requires us to have trust in hardware and standard cryptographic primitives (which are used by SGX to protect data), while a 2P-SFE protocol needs only the latter. Moving the "trust" from software to hardware presents additional problems – the authors are unaware of any techniques that could be used to sign and verify hardware. Given the recent issues with nation states actively infiltrating hardware

vendors at massive scales for bulk data collection, this is a major problem. Ultimately, the trust in SGX boils down to trust in hardware suppliers and whether or not the hardware can be opened and the CPU read.

6.4 Using SGX for 2P-SFE Computations

Having outlined the capabilities and limitations of SGX-supported 2P-SFE, we now present our solutions to the problems faced when trying to use SGX for 2P-SFE protocols.

6.4.1 Using SGX for 2P-SFE: Problems and solutions

Our solution is to augment the SGX programs to prevent (or reduce) data leakage in SGX for 2P-SFE computations. These augmentations are described below.

Timing Side Channel: We must ensure all code-paths take approximately the same amount of time. There are many such obfuscation-based palliative mechanisms, as well as general mitigation strategies [95]. However, these problems are more complex in some scenarios – *e.g.*, when a secret variable determines how many times a loop executes. In this case, the time the program takes can reveal information about the value of the secret variable. It is possible to prevent any secret values from being revealed by having a fixed loop bound, however, this may not always be preferable. We can limit the amount of information leaked when executing a loop by including a pseudo-random N extra loop iterations - where N is based upon secret information from both parties. Using this technique, neither party learns the actual amount of iterations that were executed.

Memory Side Channel: We must ensure all memory that can be touched by the SGX program is touched one single time at the beginning of the program. Once the SGX program touches a piece of plaintext memory, the memory should not be

read again unless the read is not dependent on secret information. If the read is dependent on secret information, the evaluator may be able to learn something about the secret [96, 97]. However, if we need too much data and some are encrypted and stored outside of the enclave, there might be a correlation between when a block of memory is read and when a block of encrypted memory is sent back to RAM; *e.g.*, if a binary search program that runs inside an enclave reads one element at a time, mere observation yields the secret query (within a range, if it is missing). In order to prevent this problem, we must ensure a *mix* operation is performed that removes any correlation between plaintext memory and encrypted memory; *e.g.*, this would occur if the memory was placed outside of the enclave in the same order as it was entered. Such *mix* operations, which continuously shuffle and re-encrypt data as they are accessed, already exist, and are widely used to implement Oblivious RAM [92, 98].

Array Out of Bounds: To mitigate the risk of arrays out of bounds in SGX, we apply safe memory access techniques to ensure memory does not go out of bounds. SGX programs can use bound-checking data structures or memory safe languages [90]. Although such techniques slow down the execution time of the application, both of the aforementioned methods for preventing arrays from going out of bounds would still be significantly faster than executing the programs in a 2P-SFE protocol.

Cost of a 2P-SFE protocol vs SGX:

In Table 6.1, we note the expected cost of normal 2P-SFE using garbled circuits and SGX-supported 2P-SFE. We examine the costs of setup, input, the operation itself, data access, and memory access. As shown in the table, the primary reason for the expected increase in the speed of SGX-supported 2P-SFE over a garbled-circuit protocol is the amount of cryptography required for each operation and data access in 2P-SFE (which is free in SGX). However, unlike garbled-circuit protocols, SGX encounters a cost to push memory out of the cache to RAM (*Non-Cache Access*).

	2P-SFE$_{Semi}$		2P-SFE$_{Malicious}$		SGX	
	Sym	Asym	Sym	Asym	Sym	Asym
Setup	-	-	-	-	$O(1)$	$O(1)$
Input	$O(N)$	$O(K)$	$O(N*S)$	$O(K*S)$	$O(N)^{+}$	-
Per Operation[1]	$O(1)$	-	$O(S)$	-	-	-
Data (array) Access	$O(N)$	-	$O(N*S)$	-	-	-
Non-Cache Access[2]	-	-	-	-	$O(1)$	-

Table 6.1: This table shows the cost (in terms of cryptography) for different operations in 2P-SFE and SGX-supported 2P-SFE. " - " indicates there is no cryptography required for the operation. N is length of input. C is length of the circuit/program. K is the bit-security parameter. S is the stat parameter (number of circuits in 2P-SFE). [1] - For 2P-SFE this is per gate and for SGX-supported 2P-SFE this is per processor instruction. [2] - the cost of saving and loading a value to or from main memory for SGX. $^{+}$ - assumes we attained a symmetric key during the setup phase and used it to encrypt the input.

6.4.2 Half and Half

With the techniques above, both 2P-SFE protocols and SGX can be used together in scenarios where the parties trust each other enough to want to cooperate in the first place but not enough to release private data or blindly trust the other parties not to cheat [19]. However, when different groups of parties want to perform a secure computation together, a user may trust one group over another; the different guarantees and characteristics of SGX-supported 2P-SFE and current 2P-SFE protocols mean that it might make sense to use one technique for a certain group and not for another. In this section, we examine how to perform a secure computation where one part of it is evaluated using current 2P-SFE protocols and the other is evaluated using SGX-supported 2P-SFE.

We start with two companies, A and B (as shown in figure 6.1), which want to perform a secure computation that involves nodes both inside and outside (*i.e.*, nodes belonging to the other company) their private networks. In our example, parts of the computation are done entirely inside of each company, while others require A and B to cooperate. In this way, companies could use the trust model of SGX when they

trust their own employees (and Intel) and 2P-SFE when they require trust based on cryptographic guarantees instead of assuming another company has not broken into the hardware. This idea is shown in Figure 6.1. (Another similar use case is that of two nation states that want to perform a secure computation that involves nodes both inside and outside their territories.)

To perform such hierarchical or "mixed" SGX computations, involving 2P-SFE (garbled-circuit) protocols in some sections and SGX in others, users need to know how to convert a value from a 2P-SFE protocol to an SGX-supported 2P-SFE value and vice-versa. Once we know how to perform these transformations, we can run "mixtures" of 2P-SFE protocols and SGX. For simplicity, we deal with the semi-honest setting, although we note there are ways to do the same conversions in the malicious setting. For the purposes of this short protocol, the evaluator is the evaluator in both 2P-SFE and SGX-supported 2P-SFE. The generator is the generator for 2P-SFE and the sender in SGX-supported 2P-SFE.

Before we briefly describe the conversion process, we describe more about garbled circuits. During the evaluation of the garbled circuit each wire holds an encrypted value. The generator knows the possible encrypted values (that is, which values represents 0 and 1), but does not know which value is actually on the wire (the value the evaluator has). The evaluator knows the encrypted value on each wire value, but does not know what any value represents. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity.

Conversion from Garbled Circuit to SGX:

1. For each garbled wire w_i we will convert to an SGX value, the evaluator has w_i^r (the encrypted result) and the generator has w_i^0 and w_i^1 (the encrypted values that represent 0 and 1).
2. The generator enters in w_i^0 and w_i^1 into $prog_{sgx}$ (the SGX program) as his input.
3. The evaluator enters in w_i^r into $prog_{sgx}$ as her input.

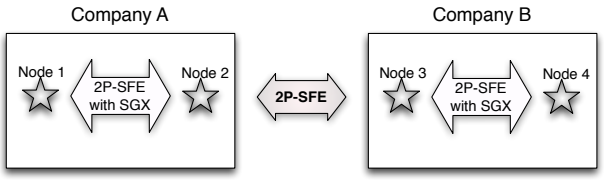


Figure 6.1: Half and Half. In this usage, we convert SGX-supported 2P-SFE values to standard 2P-SFE values and back in order to take advantage of the speed of the combined form when the trust model is acceptable and still allow for a stronger model when the trust model of SGX-supported 2P-SFE is not acceptable (say, the user does not trust Intel when using a public network).

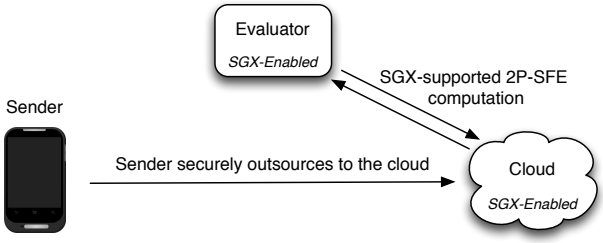


Figure 6.2: Outsourcing. Shows the different parties in our outsourcing protocol.

4. $prog_{sgx}$ calculates whether w_i^r is w_i^0 or w_i^1 as sets the corresponding input, b_i, to match w_i^r.
5. $prog_{sgx}$ uses each b_i as input.

Conversion from SGX to Garbled Circuit:

1. For each bit b_i that will be converted into a garbled value w_i, the generator creates both possible garbled values, w_i^0 and w_i^1, that will represent the two possible values of b and enters them into $prog_{sgx}$.
2. $prog_{sgx}$, based on whether b_i is a 0 or 1, selects either w_i^0 or w_i^1 to be w_i^r.
3. Each w_i^r is sent to the evaluator to be used as input to the garbled circuit.
4. The generator uses his values, w_i^0 and w_i^1, in the creation of the garbled circuit to ensure w_i^r will map to a value.

Security: In order for either the generator or evaluator to learn additional information, they have to (1) posses either w_i^0 or w_i^1 and posses w_i^r, or (2) see b_i outside of the enclave. Since b_i only exists inside of the enclave, it will not be seen by either the generator or evaluator. The generator only ever sees w_i^0 and w_i^1 and never sees w_i^r.

Likewise, the evaluator only sees w_i^r and never sees w_i^0 or w_i^1. Thus, neither party will learn any additional information.

6.4.3 Outsourcing

Many 2P-SFE computations involve devices that have computational and hardware constraints, such as not possessing an SGX unit (*e.g.*, a smartphone). For such devices, it would be very useful to have the ability to securely outsource computation to a more powerful or better equipped system. There have already been a number of works addressing this situation in 2P-SFE [20, 36, 37, 38]. In this section, we examine how we can outsource from a constrained device (that does not possess an SGX module) when we want to perform SGX-supported 2P-SFE.

In our setup, seen in Figure 6.2, the sender does not have an SGX unit and is outsourcing to a server, the cloud, who has an SGX unit. Any outsourcing protocol must guarantee (1) the party we are outsourcing to (the cloud) cannot cheat, and (2) the party that performs the SGX execution (the other party in the original SGX-supported 2P-SFE computation, the evaluator) cannot cheat.

We assume that we are trying to protect the input and output of the sender; we also assume the cloud and evaluator do not collude, *i.e.*, they are not working together to corrupt the sender's output or input. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity. As before, superscripted "+" and "−" signs denote public and private keys, respectively.

Protocol:

1. The cloud and evaluator perform the standard SGX setup to initialize their SGX units and confirm they are running the desired program.
2. Both parties pass enclave public keys, $Ecv^+_{key_{cloud}}$ and $Ecv^+_{key_{eval}}$ to the sender and authenticate by using MRSIGNER [87, 88].
3. Both the evaluator and cloud enclaves send to the sender their enclave measure-

ments, $Ecv^{cloud}_{measure}$ and $Ecv^{eval}_{measure}$.

4. The sender checks the measurements $Ecv^{cloud}_{measure}$ and $Ecv^{eval}_{measure}$ are correct.
5. The sender encrypts his input, $input_{sender}$, and a public key for his output, Out^{+}_{key}, under $Ecv^{+}_{key_{eval}}$ to create $Enc(input_{sender}||Out^{+}_{key})$ and sends it to the cloud.
6. The cloud enters $Enc(input_{sender}||Out^{+}_{key})$ into the SGX program, $prog_{sgx}$. *We note here there is no reason the cloud cannot also have input into the program, if it is desired.*
7. The input is sent from the cloud to the evaluator.
8. $prog_{sgx}$ is executed according to the previous SGX-supported 2P-SFE protocol.
9. The sender's output, $output_{sender}$ is encrypted under Out^{+}_{key} as a final step in $prog_{sgx}$.
10. The sender's output $Enc(output_{sender})$ is sent from the evaluator to the sender.
11. The sender uses the output private key Out^{-}_{key} to decrypt $Enc(output_{sender})$.

Security of the Sender's Data

Input: Since the sender's input is encrypted under the evaluator's enclave private key, it can only be decrypted inside of the evaluator's enclave. Given the measurement of the evaluator's enclave, we also know the program inside of the enclave is correct so it will not pass the input outside the enclave.

Output: Since the sender's output is encrypted inside the enclave during evaluation and is only sent outside when it is encrypted under the sender's public key, only the sender can decrypt and read this output.

6.4.4 Improving the security of 2P-SFE protocols using SGX

Semi-honest or honest-but-curious (HBC) protocols guarantee security as long as all participants faithfully follow the protocol, but the participants can attempt to glean extra information using all the messages they see. Semi-honest protocols are much cheaper in terms of computation cost than those that protect against malicious adversaries, who attempt to gain additional information by any means necessary. We

can use SGX for parts of the semi-honest 2P-SFE protocol to gain additional security guarantees without incurring significant overhead. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity.

First, we replace the OT in the 2P-SFE protocol with an SGX component that acts like an OT. The SGX OT is a stripped down version of the previously described SGX-supported 2P-SFE protocol. In this program, the 2P-SFE evaluator chooses the encrypted form of the input as in the 2P-SFE protocol. This immediately gives us greater security than the standard semi-honest OT since we are not relying on the parties to behave correctly during the OT (i.e., the SGX unit checks whether the parties are running the correct "OT" program). Note that this does not guarantee fair-release of the result, since a malicious party can still cause us to abort at any point.

Similarly, we can replace the circuit generation and evaluation with an SGX component as well. This SGX-evaluation is the program evaluation component described earlier. While we could use the 2P-SFE OT before this part of the protocol, using the SGX OT component gives us better security. After the input and circuit evaluation components are replaced, we can also replace the output component with the SGX output protocol. Replacing all of these elements leaves us with a protocol that is significantly more secure than the original semi-honest 2P-SFE protocol (since the SGX protocol has checks for when a user is malicious and the HBC protocol does not), while remaining much cheaper than a malicious 2P-SFE protocol.

6.4.5 Universal Programs (Circuits)

A *universal circuit* (UC) is a program that takes another program as input (denoted as UC_{prog}) and then executes it. In a UC for two parties, one party enters UC_{prog} as input while the other party enters the input for UC_{prog}. For example, this can be used to protect secret hashing algorithms: one party enters the secret cryptographic

hash algorithm and the other party enters a value to be hashed. Using a UC, the cryptographic hash algorithm will be kept secret while the hash is computed.

However, in 2P-SFE, a UC requires a massive number of array accesses due to the nature of oblivious data access. For each operation in UC_{prog} (*e.g.*, $data[a] = data[b] + data[c]$), the inputs to the operation (*i.e.*, $data[b]$ and $data[c]$) have to be found from all the possible values that could be entered into the instructions – *i.e.* this requires a set of if statements to check whether index value v equals b – unless constraints can be added to UC_{prog}. However, in SGX-supported 2P-SFE, this would be more efficient since array access takes $O(1)$. Thus, UC programs can be efficiently executed in an enclave.

6.4.6 Novel Use Cases for SGX

Secure data storage: The security of data storage is a growing problem in the world today. With the advent of cloud and multi-user systems, it is often the case that users want to break polices to gain access to other users' data. SGX could be used as a gatekeeper to prevent unauthorized users from accessing the stored data. If all reads and writes went through the SGX hardware, it would be possible to automatically encrypt and decrypt it, based on a user-entered key, without the need for a specialized drive. A keyboard could enter the password to the enclave while skipping the operating system and any keyloggers within. Unlike systems such as BitLocker [99, 100], the key here would remain safe even if the operating system was compromised. Such a design would also work for cloud storage, where the SGX program would automatically encrypt data before they are sent to the cloud server; it could be implemented so as to be transparent to the end user and obviate the need to trust cloud companies.

User Authentication: SGX offers many new avenues for user authentication. It includes MRSIGNER, which signs the enclave before it is deployed. Group authenti-

cation is also possible, using EPID (Enhanced Privacy ID) [87], which is an extension to the Direct Anonymous attestation scheme used in [101,102]. This allows an enclave to sign communications while maintaining privacy within a group – *i.e.*, an entity can verify that *some* member of the group signed the communication, but cannot identify *which*. There is also a "pseudonymous" mode, which relaxes the security slightly, allowing the verifier to know whether it has checked an enclave in the past while still maintaining intra-group anonymity.

Cyber-physical applications: Given the security concerns involved in control systems for sensitive infrastructure (say, a nuclear power plant or a hydroelectric dam), improving security is highly desirable. In the hydroelectric dam example, such a control system would regulate the amount of water going into the hydroelectric generators, the amount of water contained in the reservoir, and the water exiting through the spillways. In order to prevent attacks on such systems, the controls could be made accessible only through an enclave that would require all orders to the system to be signed. Another benefit to using an enclave would be to hide the current state of the system. Periodic signed updates from the enclave to a "master" control system would prevent the system from being taken offline without the knowledge of the master control system. These strategies would mitigate the threat of hackers breaking into the system and altering code or stealing passwords – this information would exist only inside of the enclaves.

Online Games: Online games are played between multiple users on different machines. In order to prevent a massive amount of bandwidth from being transferred to and from the users machines, many games only pass events from one user to another, *e.g.*, the information for each user command. Each local machine is then able to process this information independently. This comes at the cost of each local machine knowing the entirety of the game's data, including sensitive information about other player's positions. SGX could be used to protect each gamer's private

data from other gamers by performing some of the computation inside of the enclave. By keeping information about each gamer's own strength inside an enclave, a hacker (or any user who uses a tool to read information normally not available to them) would be unable to gain any private information. The enclave would release such private information to the local machine based on triggers in the code, *e.g.*, when an enemy unit is nearby. Furthermore, we can periodically verify the state of each enclave to prevent cheating. Using this concept, players can store data other belonging to users on their machines (so they don't need large amounts of network bandwidth), but they cannot read any private data, which are safely stored within an enclave.

6.5 Results

Currently, SGX is not implemented on any commercially available hardware. As part of this work, we attempted to use OpenSGX to demonstrate the practical ramifications of using SGX-supported 2P-SFE. We were unable, however, to get OpenSGX to reliably perform attestations of remote enclaves. Nonetheless, we present results from a universal circuit program run on OpenSGX and additional approximations. All the tests were run on an Alienware Aurora-ALX R4 2012 with 32Gig RAM and 3TB HDD.

Universal Circuit: We created universal circuit programs for both a highly optimized variant of the Kreuter *et al.* implementation of Yao's garbled circuits [3] and SGX-supported 2P-SFE using OpenSGX. Each program runs 20,000 gates with 100 inputs, 100 outputs, where each gate output could be stored in 600 possible places. The Yao program had 360,760,002 non-XOR gates (XOR gates require no cryptographic operations or bandwidth [40]) and took 842.9 seconds (average over 10 runs) to execute in a semi-honest implementation. In contrast, the SGX-supported 2P-SFE version ran in about 1 second, giving us an improvement of almost 3 orders

	2P-SFE$_{Semi}$		2P-SFE$_{Malicious}$		SGX	
	Sym	Asym	Sym	Asym	Sym	Asym
Hamming160	792	80	63,360	6,400	$3 + M$	3
RSA512	6,460,281,864	80	516,822,549,120	6,400	$12 + M$	3
MatrixMult16	16,753,664	80	1,340,293,120	6,400	$128 + M$	3
Mult4096	134,238,216	80	10,739,057,280	6,400	$96 + M$	3

Table 6.2: This table shows approximate amount of cryptography required for both 2P-SFE and SGX-supported 2P-SFE. Symmetric operations are for garbling each gate and input operations (2P-SFE). Asymmetric operations are for OTs (2P-SFE) and signing measurements (SGX). M is the measurement operation; it uses AES-128 while it MACs the SGX program and takes $length(program)/128$ symmetric operations.

of magnitude.

Approximations: We present approximate results, in terms of symmetric and asymmetric cryptographic operations, for four programs in Table 6.2. It is clear from the values in the table that the number of operations is substantially smaller under SGX than under a 2P-SFE protocol.

Programs: (1) A 512-bit RSA program, which performs a single modular exponentiation where m, e, and P are of 512 bits in length. (2) A 160-bit hamming distance program, which computes the hamming distance of two 160 bit numbers. (3) Matrix multiplication with two matrices of 16x16 integers that produces a third integer matrix of 16x16 values. (4) Multiplication of two 4096-bit integers to produce a 8192-bit integer. The approximations presented in Table 6.2 were calculated as follows:

- **HBC Symmetric:** (Number of gates in the program)×4 + (input length).
- **HBC Asymmetric:** 80 operations (for the Ishai OT extension [25] set to 80-bit security).
- **Malicious Symmetric:** (Number of gates in the program)×(circuits) + (input length)×(circuits).
- **Malicious Asymmetric:** 80×(circuit operations for the cut-and-choose Ishai OT extension secure against malicious adversaries).
- **2P-SFE w/SGX Symmetric:** (input length) / (128) + (output length) / (128) + (measurement of program). Using "128" assumes we use a 128-bit algorithm

(such as 128-bit AES) for the input and output encryption. We do not attempt to compute the exact number of operations for the measurement, but this should be the length of the program divided by the bit-length of the MAC function [87].

- **2P-SFE w/SGX Asymmetric:** 3 operations: 1 to sign the measurement, 1 to check the other enclave's measurement, 1 to sign (or check the signature) of the output.

6.6 Previous Work on Secure-Execution Environments

In this section, we briefly discuss previous work on the use of specialized software and hardware platforms to enable secure execution of code. However, none of these works provide the same guarantees or address the same scenarios as a 2P-SFE protocol.

Various levels of code and data protection have been achieved in the past using approaches as varied as managed runtime environments (such as Java and .NET), tamper resistant software [103], and microkernels.

Haven [89] is an SGX-based system for executing Windows applications in the cloud. VC3 [90], also based on SGX, allows verifiable and confidential execution of MapReduce jobs in untrusted cloud environments. Our work is significantly more secure than Haven, and can withstand controlled-channel attacks. (An extension of this work with experiments is in progress, but cannot be currently included due to non-disclosure agreements with Intel.)

Systems such as TrustedDB [104] and Cipherbase [105] use different kinds of trusted hardware to process database queries over encrypted data. There exist several other systems [106, 107, 108] that use trusted system software (usually a trusted hypervisor) along with specialized hardware to achieve various security and privacy requirements. Some, such as Virtual Ghost [109] and Flicker [110], avoid hypervisors

by using specialized kernel-level hardware-isolation mechanisms and time-partitioning between trusted and untrusted operations, respectively. Super-distribution systems for transmission of protected digital data also exist [111]. They decrypt protected data using a key from an authorized clearinghouse and then re-encrypt the data with a locally generated key on the end-user system, ensuring that no one else can use the data. Secure co-processors [112] allow programs to execute securely as long as users can verify that they are dealing with untampered programs and hardware. None of these, however, has attempted to combine SFE with hardware to produce faster or more secure protocols.

Intel has a number of whitepapers on SGX [87,88], as well as previous attempts in the same vein, such as the Trusted Execution Technology [113]. ARM trustzone for Cortex-A processors also provides some similar guarantees and has been used to build embedded linux platforms [114], language runtimes for mobile applications [115], and many other systems.

Chapter 7

The SysSC-UI Tool

7.1 Introduction

Given the huge and growing body of knowledge in SMPC and the myriad protocols with varying security assumptions and guarantees, it is extremely difficult even for experienced researchers, let alone everyday users or businesses, to decide which technologies to employ in order to solve a particular problem.

In this chapter, we introduce a tool, SysSC-UI, which allows naïve users to search and specify their security needs and assumptions. We describe a set of axes along which we classify SMPC protocols; users can specify where their desired scenario lies on these axes, at which point the tool produces a list of known cryptographic protocols suited for the scenario specified or indicates that no such protocols are known. The system is designed to be easily extensible to include new axes and protocols so that other researchers can add to the body of knowledge covered by the tool. The tool is intended to:

- Allow potential users to explore the space of SMPC protocols to see whether there exist known solutions for their problem scenario. Hopefully, this will make it easier for naïve users to adopt privacy-preserving solutions based on SMPC.

- Help new researchers get up to speed in a complex area by providing a quick and easy annotated bibliography and search system.
- Facilitate new research by making it easy to find gaps in the literature – combinations of assumptions or guarantees that do not exist – which could indicate the existence of an unexplored scenario or an as-yet unproven impossibility result for that scenario.

To the best of our knowledge, there are no other tools that serve the same functions as SysSC-UI. There do exist a few review papers and surveys of SMPC protocols [116, 117, 118, 119], but they all restrict themselves to specific security models or problem scenarios. There also exist a number of papers which discuss the real-world adoption of SMPC, such as the SMPC-in-the-field experiments of Feigenbaum *et al.* [120] and Bogetoft *et al.* [15] and the end-user survey work of Kamm *et al.* [121].

The rest of the chapter is organized as follows: Section 7.2 describes the axes we use to classify SMPC protocols, which form the foundation of the tool. Section 7.3 discusses the construction of the database the tool uses to search for protocols satisfying a given problem scenario. Section 7.4 describes the user interface.

The material in this chapter appeared in preliminary form in [23].

7.2 The Axes of Classification

In order for us to be able to specify various problem scenarios, we need to first have a way to quantify SMPC protocols.We do this by creating a set of linear axes, each representing an important feature, assumption, or guarantee of the surveyed protocol. Every axis has at least two labeled values, at the endpoints, and may be continuous or discrete. We score SMPC protocols along these axes, allowing us to compare them in an easy and intuitive way. To the best of our knowledge, this is the first attempt to classify the field in this fashion.

We selected the axes to be as orthogonal as possible, minimizing overlap (although some logical dependencies between axes are unavoidable). We also made sure that the set of axes was *complete*, so that they could express all distinctions of security and (asymptotic) efficiency between any two protocols.

As a necessary prerequisite for this work, we carried out an extensive literature survey producing an annotated bibliography of SMPC research [122]. This survey comprised over 180 papers from the SMPC literature as well as a number of important papers on the primitives that underlie the majority of SMPC protocols, such as oblivious transfer.

The annotated bibliography includes a short description of the paper along with tags describing the essential characteristics of the work, *e.g.*, `2party` for a two-party protocol, or `uncond` for a protocol with unconditional security. These tags make it easy to write simple scripts that can generate lists of papers for a particular setup.

7.2.1 Linear representations of SMPC protocol features

In the diagrams below, we describe the axes and the intermediate points that can be selected using the tool to specify various problem scenarios. We emphasize that these points represent the current state of research and are not intended to be static; the tool has been specifically designed to make it very easy to add new points to the axes. Such events have occurred in the past, such as the introduction of fail-stop and covert adversaries by Aumann *et al.* [123] and Goyal *et al.* [30] which added new intermediate points on axis 7 ("Maliciousness").

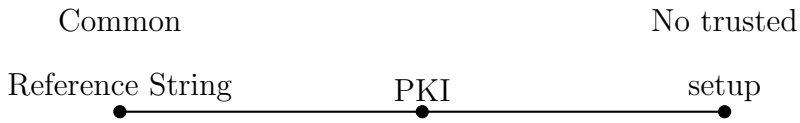
The axes are oriented so that moving from left to right results in an "improved" protocol, *i.e.*, any point on the right represents greater efficiency, a weaker assumption, or a stronger guarantee. We do not include the model of computation among our axes, but this is present in the entry in our SMPC protocol database, described in Section 7.3. The axes are informally grouped into four categories; those in categories

I and II (*Environmental Features* and *Assumptions*) can be thought of as what one "pays" for a given protocol, and categories III and IV (*Security* and *Efficiency*) can be thought of as what one is "buying."

I. Environmental Features Axes

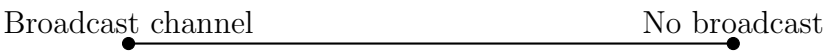
This is the category of features or primitives assumed to be provided by the execution environment. The right endpoint indicates that the feature is not required in any form.

1. Trusted setup



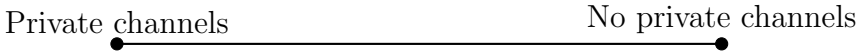
Protocols achieving the highest composable security levels require some type of trusted data to be shared by all the parties prior to the protocol execution. The middle point, PKI, is occupied by protocols such as that of Barak *et al.* [124], who showed how to use public-key-like assumptions instead of a polynomial-length common reference string in any case where computational security suffices.

2. Broadcast



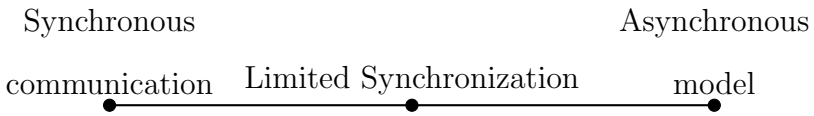
The broadcast-channel assumption means that each party has the ability to send a message to all other parties simultaneously, and that all parties receiving the broadcast have assurance that the same message was received by all parties.

3. Private channels



The private channel assumption is only significant for unconditionally secure protocols, because cryptography using basic computational-hardness assumptions can be used to emulate private and authenticated communication channels.

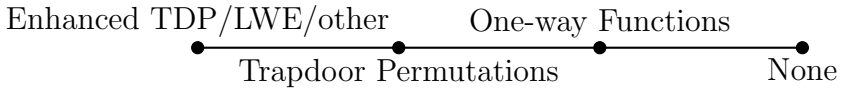
4. Synchronization



A basic assumption of the early SMPC protocols is that they operate on a *synchronous* network, in which all sent messages arrive on time and in order. The asynchronous case was first considered in Ben-Or *et al.* [125], where messages may be arbitrarily delayed and arrive in any order. Note that, in such a case, it is impossible to know whether a corrupted party has failed to send a message or, rather, the message is simply delayed. Later works, such as that of Damgård *et al.* [12], have staked out intermediate points on this axis by giving protocols that require a smaller number of synchronization points (typically a single one).

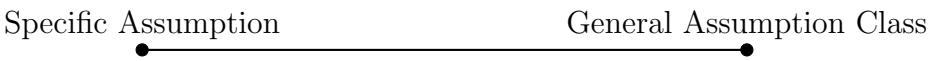
II. Assumption Axes

5. Assumption level



A total (linear) ordering for cryptographic assumptions is not known, and the separation of assumptions cannot currently be unconditionally proven. We therefore use broader categories of assumption type, because these are usually sufficient to distinguish protocols. If a protocol makes no such assumptions, it is said to have *unconditional* security (see Axis 10). Some work specifies protocols in a *hybrid* model, with no concrete computational assumptions, but in which some high-level cryptographic operation (such as oblivious transfer) is assumed to exist as a black box.

6. Specific or general assumption

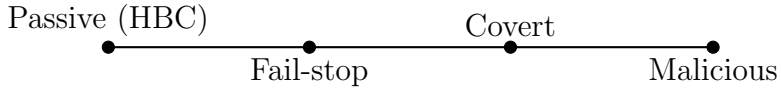


Some more efficient protocols have been designed by making use of specific number-

theoretic assumptions. This axis indicates whether the protocol requires such assumptions or whether it is stated so as to use any assumption from a given class, *e.g.*, trapdoor permutations.

III. Security Axes

7. Adversary maliciousness



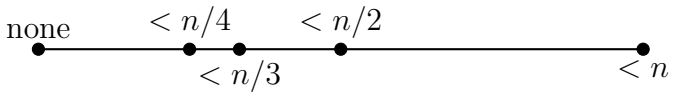
A passive, or honest-but-curious, adversary is one that follows the protocol but may use the data of corrupted parties to attempt to break the protocol's privacy. A fail-stop adversary follows the protocol except for the possibility of aborting. A malicious adversary is one whose behavior is arbitrary, and a covert adversary is like a malicious one, except that it only deviates from the protocol if the probability of being caught is low.

Not present on the axis is the value "rational," since the class of rational adversaries is in fact a generalization that can encompass the entire axis, except for fully malicious, because malicious behavior can be truly arbitrary. The position of a rational adversary on the axis is determined by its utility function.

8. Adversary mobility

A static adversary must choose which parties to corrupt before the protocol begins. An adaptive adversary can choose which parties to corrupt, up to the security threshold (see Axis 9), over the course of the computation, after observing the state of previously corrupted parties. A mobile adversary is able to move corruptions from one party to another in the course of the computation.

9. Number of corrupted parties tolerated



This is the maximum number of corrupted parties for which the (strongest) security guarantees of the protocol hold. The values shown are chosen merely to be representative of the most well known protocols; any value along the axis is possible.

Some protocols tolerate additional corrupted parties at a lower level of maliciousness; see axes 13 and 14.

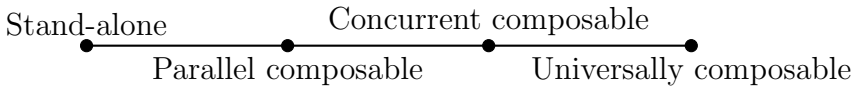
10. Security type

Both statistical and perfect security are unconditional, that is, not based on computational hardness assumptions. Note that true unconditional security typically cannot be achieved through the internet, even if an unconditionally-secure protocol is used, since all unconditionally secure protocols require the assumption of either private or broadcast channels, which on the internet must be emulated by cryptography.

11. Fairness guarantee

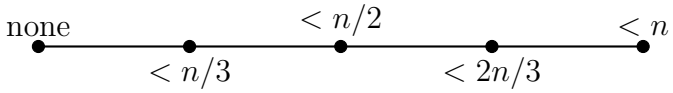
A protocol is *fair* if all honest parties receive the output if any party does. Agreement means that either all honest parties receive the output or none of them do. Protocols without agreement were introduced by Goldwasser *et al.* [126]. Some authors use the term "with abort" to refer to the no-fairness situation, in which dishonest parties can abort after receiving the correct output.

12. Composability



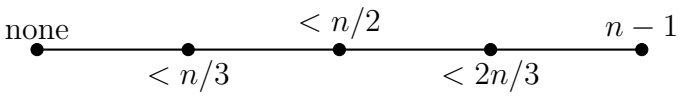
The composability guarantees of a protocol indicate whether that protocol remains secure when executed in an environment where other protocols may be executed sequentially or in parallel. The strongest guarantee, *universal composability* (UC), implies that the security properties of a protocol hold regardless of the environment in which it is executed.

13. Bound for additional passively corrupted parties tolerated



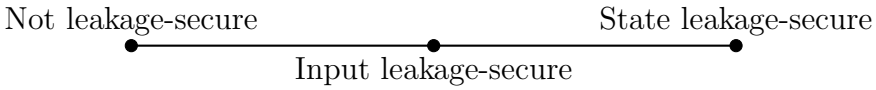
This axis applies to protocols achieving "mixed adversary" security. A protocol that tolerates a certain proportion of maliciously/covertly corrupted parties may also tolerate an additional number of passively corrupted parties, up to a certain threshold. The values on this axis represent that upper threshold. This and the following axis relate to SMPC protocols with "graceful degradation", which is surveyed in Hirt *et al.* [127].

14. Corrupted parties tolerated with weakened security



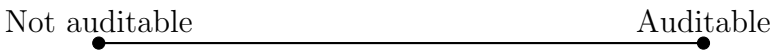
This axis applies to protocols with "hybrid security" results. A protocol that tolerates a certain proportion of corrupted parties (Axis 9) may in fact tolerate a larger number of corruptions, but with a weaker security type, *e.g.*, computational vs. unconditional security.

15. Leakage Security



Leakage security is an additional guarantee that an adversary cannot gain an advantage even if it can force all honest players to "leak" some bits of information about their state in the course of the computation. See definitions in Bitansky *et al.* [128].

16. Auditability

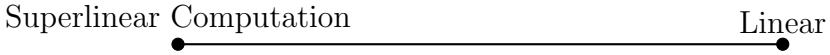


This axis indicates whether the protocol includes computations that allow for examining the transcript of computation after it is finished, to prove that the parties have correctly followed the protocol. This may be the most recent axis to come into existence, starting with the work of Baum *et al.* [129]

IV. Efficiency Axes

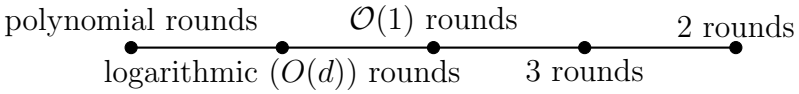
Our efficiency axes are concerned primarily with the asymptotic efficiency of the protocol in question.

17. Online computational overhead



Historically, the main efficiency concern in SMPC has been with communication rather than computational complexity; thus the lack of elaboration of this axis. More recently, Ishai *et al.* have notably shown how to achieve SMPC with constant computational overhead [130], and the RAM-model results of Gordon *et. al* [131] have shown the possibility, in the RAM model, of SMPC with amortized computation that is sublinear in the input size.

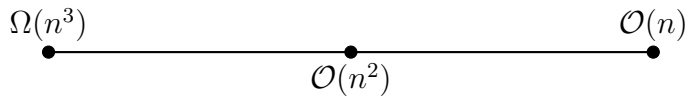
18. Online communication complexity (rounds)



Here, d is the depth of the circuit representing the functionality. Minimizing the number of rounds of computation, independently of the total amount of bytes

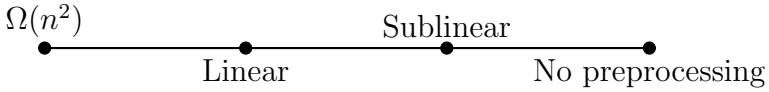
communicated, is crucial for efficiency in a high-latency or asynchronous network environment. Fully general SMPC was shown to require at least three rounds in Gennaro *et al.* [132], although for some functionalities a 2-round protocol is possible.

19. Online communication complexity (per-gate)



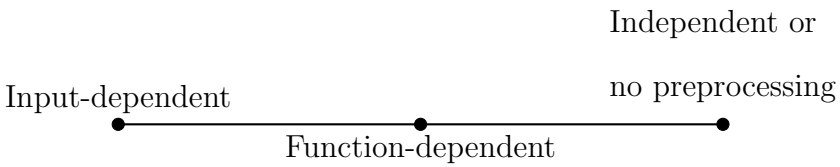
This is the most significant measure of efficiency for SMPC protocols. It can represent either bits or field elements of communication. The original BGW protocol has a communication complexity of $O(n^6)$ bits per multiplication gate in the worst case. Anything cubic or worse occupies the lowest position on our axis, as finer distinctions at that level would have little value for distinguishing current, more practical protocols.

20. Preprocessing Communication complexity



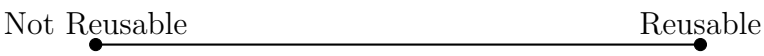
Many recent protocols achieve improved online communication efficiency by means of a preprocessing phase. In the case where the functionality is represented as an arithmetic circuit, the preprocessing phase is typically a simulation of a trusted dealer that distributes *multiplication triples*, which allow local evaluation of multiplication gates in the online phase. Sublinear preprocessing typically indicates that the preprocessing consists only of exchange of public keys, which is also indicated as a setup assumption (Axis 1).

21. Preprocessing Dependency



In some protocols, the preprocessing phase depends on the specific functionality to be computed, while in others it only depends on the upper bound of the size of the circuit. In all cases, the preprocessing is independent of the parties' inputs.

22. Preprocessing Reuse



This indicates whether the information computed in the preprocessing stage, of whatever type or amount, can be reused for multiple computations. Data of the nature of a public key typically can be reused, while *e.g.,* garbled circuits traditionally cannot be reused without breaking security. But see recent work of Goldwasser *et al.* [50].

7.2.2 Dependencies

While we made every attempt to ensure that the axes we selected were orthogonal and independent, there are some inherent tradeoffs and known theorems about relationships between our axes. In this section, we briefly cite and describe these constraints. When one of the points described below is selected on the tool, it immediately disqualifies various points on other axes. We emphasize that is specifically for cases where the combination of those points is either inherently absurd or there exists a known impossibility result, and not for cases where there is simply no known protocol. These constraints can easily be represented in a programming or knowledge representation language, as shown in Section 7.4.

Note that these theorems are easily expressible in terms of our chosen axes, further validating our selections. We now list these constraints, each accompanied by a statement and a reference to the paper where the constraint was proven.

Theorem 1 ([133])**.** *If statistical or perfect security is obtained, then either a broadcast channel or private channels must be assumed.* ***Axis constraint****: If Axis 10's*

value is to the right of "Computational," then either Axis 3's value is "Private channel" or Axis 2's value is "Broadcast channel".

Theorem 2 ([134]). *No protocol with security against malicious adversaries can tolerate more than $n/2$ corrupted parties without losing the complete fairness property.* ***Axis constraint****: If Axis 7's value is "Malicious" and Axis 9's value is to the right of $n/2$, then Axis 11's value must be to the left of "Complete fairness".*

Theorem 3 ([135]). *No protocol unconditionally secure against malicious adversaries can guarantee output delivery with $n/3$ or more corrupted parties.* ***Axis constraint****: If Axis 7's value is "Malicious," Axis 10's value is to the right of "Computational," and Axis 9's value is to the right of "$n/3$," then Axis 11's value must be to the left of "Guaranteed output".*

Theorem 4 ([133]). *No protocol can have perfect security against more than $n/3$ maliciously corrupted adversaries.* ***Axis constraint****: If Axis 7's value is "Malicious" and Axis 9's value is to the right of $n/3$, then Axis 10's value must be to the left of "Perfect".*

Theorem 5 ([132]). *Any general SMPC protocol with complete fairness against a malicious adversary must have at least three rounds.* ***Axis constraint****: If Axis 7's value is "Malicious" and Axis 11's value is at or to the right of "complete fairness", then Axis 18's value must be to the left of "2 rounds".*

Theorem 6 ([133]). *For unconditional security against t maliciously corrupted players, $n/3 \leq t < n/2$, a broadcast channel is required.* ***Axis constraint****: If Axis 10's value is to the right of "Computational" and Axis 7's value is "Malicious" and Axis 9's value is to the right of $n/3$, then Axis 2's value must be "Broadcast channel".*

Theorem 7 ([136]). *For (even cryptographic) security against $\geq n/3$ maliciously corrupted players, either a trusted key setup or a broadcast channel is required.* ***Axis***

***constraint**: If axis 7's value is "Malicious" and Axis 9's value is to the right of $n/3$, then either Axis 2's value must be "Broadcast channel," or else Axis 1's value is to the left of "No trusted setup."*

Theorem 8 ([133]). *There can be no unconditionally secure protocol against an adversary controlling a majority of parties.* ***Axis constraint**: Axis 10's value can be to the right of "Computational" only if Axis 9's value is at or to the left of $n/2$.*

Theorem 9 ([137]). *There is no protocol with UC security against a dishonest majority without setup assumptions.* ***Axis constraint**: If Axis 9's value is to the right of $n/2$ and Axis 12's value is "Universally composable," then axis 1's value must be to the left of "No trusted setup".*

Theorem 10 ([125]). *In an asynchronous environment, there is no protocol with guaranteed output secure against a fail-stop adversary corrupting $n/3$ or more parties.* ***Axis constraint**: If Axis 4's value is "Asynchronous," Axis 7's value is at or to the right of "Fail-stop," and Axis 11's value is at "Guaranteed output," then Axis 9's value must be at or to the left of $n/3$.*

Theorem 11 ([125]). *In an asynchronous environment, there is no protocol with guaranteed output secure against a malicious adversary corrupting $n/4$ or more parties.* ***Axis constraint**: If Axis 4's value is "Asynchronous," Axis 7's value is "Malicious," and Axis 11's value is at "Guaranteed output," then Axis 9's value must be at or to the left of $n/4$.*

7.3 An Extensible Protocol Database

After we completed our survey, we compiled the most significant results into a protocol database. Many papers include multiple protocols; we created a separate entry for each such protocol.

Efficiency. Our efficiency axes are based on asymptotic efficiency. We note that the model of computation in which a protocol's functionalities are expressed can have a large impact on concrete efficiency. The original Yao model uses Boolean circuits, while many of the current SMPC protocols use arithmetic-circuits. Implementing these requires performing field arithmetic, which can have a significant impact on efficiency. This difference in concrete efficiency is not captured by the axes.

Substitutability. Many SMPC protocols use subprotocols in a black-box fashion, making it possible to substitute different protocols implementing that operation. This might alter not only the performance but also the computational and environmental assumptions and security and composability guarantees of the resultant protocol. In some cases, a new and improved subprotocol can trivially be used to improve an older SMPC protocol, but no published work explicitly presents the improvement; in other cases a protocol explicitly allows for black-box substitution of subprotocols, in which case it is said to be stated in a *hybrid* model. In the case of the *OT-hybrid* model, in which oblivious transfer is a black box, recent work in *OT extension* has produced significant performance gains. We represent only concrete instantiations of hybrid protocols. The tool requires that a protocol in a hybrid model must first be "instantiated" with concrete sub-protocols in order to be scored.

2P-SFE. While 2P-SFE can be considered a subset of general SMPC, vast improvements in efficiency have been made for the two-party case so we treat it separately. We note that these optimizations often come at the expense of *symmetry*: The security guarantee against cheating by one of the two parties may be weaker than for the other. For instance, one of the two parties may be able to cheat with an inverse-polynomial probability, while the other may only be able to cheat with negligible probability. Asymmetry is not included in our axis definitions; we score two-party by the weaker of the two sets of security guarantees.

7.4 Tool Description and Usage

In this section, we describe an an interactive tool we have developed for interacting with the SMPC protocol database based on the axes we described in Section 7.2.

SysSC-UI has a graphical user interface which reads in a protocol database of axis values and enables the user to adjust a set of sliders and checkboxes corresponding to our axes. A dynamically updated "results" window displays the protocols from the database that match the specified axis values, *i.e.*, for a given setting of the sliders and checkboxes, the results window shows all papers whose axis values are at the same level *or higher* than the settings.

Figures 7.1 and 7.2 provide screenshots of the application. The source code for the desktop version is available online `https://code.google.com/p/syssc-ui/`. The web-based version `http://work.debayangupta.com/ssc/`.

When the tool is started, the sliders and checkboxes are all set to the least constraining position, so every protocol in the database is displayed in the results window. There is a reset button to bring all the sliders back to this state; it is also possible to highlight any protocol and move the sliders to the values for that protocol. (This also updates the results to show protocols that are "at least as good" as the one selected.) Double-clicking on a reference in the results window displays more information about the paper from the annotated bibliography.

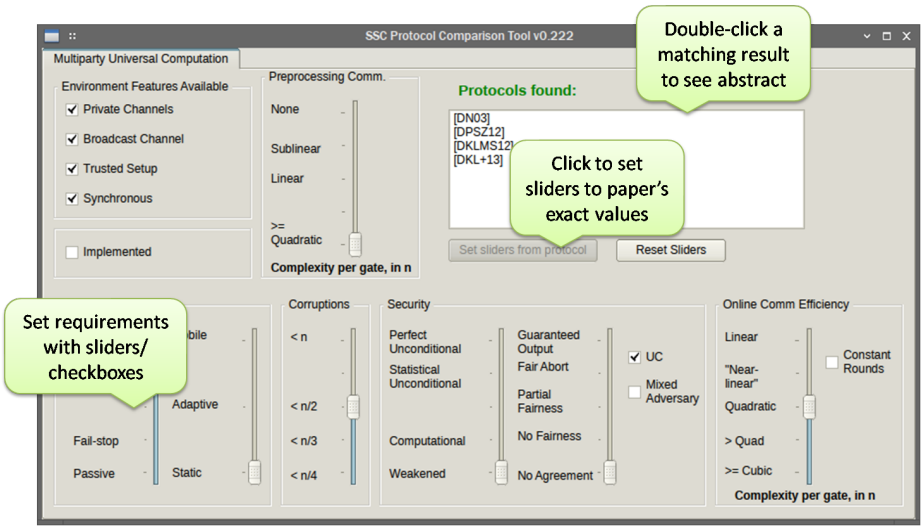


Figure 7.1: SysSC-UI tool interacting with the protocol database.

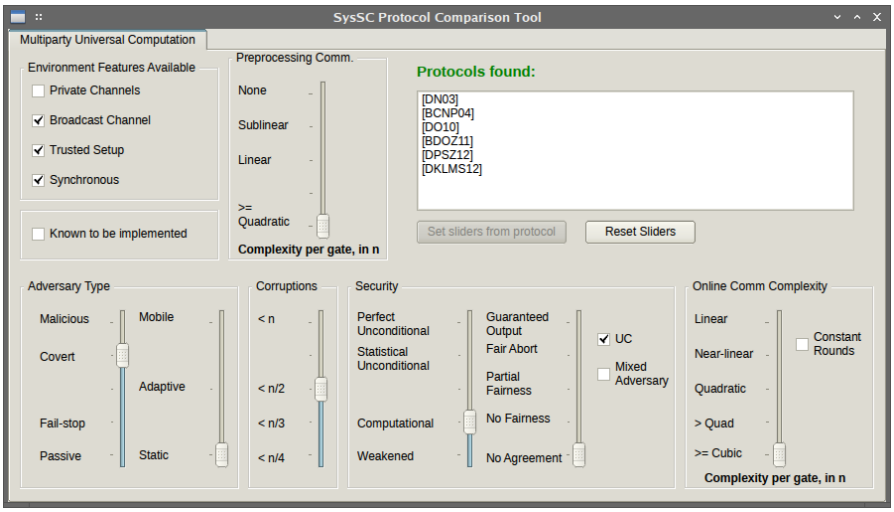


Figure 7.2: Results shown by the SysSC-UI tool after selections have been made.

Chapter 8

Conclusion

We present a number of technologies intended to make SMPC deployable in real-world situations.

Chapter 4 introduces PartialGC, a server-aided SFE scheme that allows the reuse of encrypted values, fundamentally changing the communication pattern of SFE protocols and introduces the ability to save state across multiple computations. The reduced overheads from the costs of input validation and reusing saved values reduce computation costs by up to 96% and bandwidth costs by up to 98%.

In chapter 5, we examine the state of SMPC compilers using rigorous validation testing. From this examination, we present a set of guiding principles for secure computation compiler design and develop the Frigate compiler based on these principles. Our compiler reduces compile time by as much as 447x when compared to previous circuit compilers. Furthermore, our novel circuit representation format allows for circuit interpretation time to be reduced by as much as 786x and execution time by up to 21x. These results demonstrate that a principled approach to design and validation of secure computation compilers produces tools that are both correct and efficient, and offer the community a solid foundation on which to develop further research.

Chapter 6 presents the first systematic consideration of Intel's Software Guard Ex-

tensions as a platform on which to implement two-party secure function evaluation. We show that careful use of SGX primitives can facilitate extremely efficient 2P-SFE protocols, provide an outsourcing mechanism for machines without an SGX module, and discuss augmentations to SGX, which provide stronger guarantees against leakage. We also use SGX to convert 2P-SFE protocols that are secure against semi-honest adversaries into ones that are secure against malicious adversaries, and discuss a number of use cases for SGX. As SGX-enabled processors eventually make their way onto the market, future work will include the implementation of these protocols and improvements to their efficiency and security properties.

Finally, chapter 7 introduces a tool that can significantly ease the task of coming to grips with the body of results in the SMPC and potentially speed its adoption in the real world by allowing users to easily search through a large number of known results in the area. We also present a web-based survey (available at `http://goo.gl/T4ORzr`) that allows researchers to submit descriptions of new protocols and their features on the axes so that they can be integrated into the protocol database and SysSC-UI.

Together, we hope that these software and protocols will help make SMPC feasible for use by users in the real world.

Appendices

Appendix A

Partial Garbled Circuits

A.1 CMTB Protocol

As we are building on the CMTB garbled circuit execution system, we give an abbreviated version of the protocol. In our description we refer to the generator, the cloud, and the evaluator. The cloud is the party the evaluator outsources her computation to.

Circuit generation and check The template for the garbled circuit is augmented to add one-time XOR pads on the output bits and split the evaluator's input wires per the input encoding scheme. The generator generates the necessary garbled circuits and commits to them and sends the commitments to the evaluator. The generator then commits to input labels for the evaluator's inputs.

CMTB relies on Goyal et al.'s [30] random seed check, which was implemented by Kreuter et al. [3] to combat generation of incorrect circuits. This technique uses a cut-and-choose style protocol to determine whether the generator created the correct circuits by creating and committing to many different circuits. Some of those circuits are used for evaluation, while the others are used as check circuits.

Evaluator's inputs Rather than a two-party oblivious transfer, we perform a three-party *outsourced oblivious transfer*. An outsourced oblivious transfer is an OT that gets the select bits from one party, the wire labels from another, and returns the selected wire labels to a third party. The party that selects the wire labels does not learn what the wire labels are, and the party that inputs the wire labels does not learn which wire was selected; the third party only learns the selected wire labels. In CMTB, the generator offers up wire labels, the evaluator provides the select bits, and the cloud receives the selected labels. CMTB uses the Ishai OT extension [25] to reduce the number of OTs.

CMTB uses an encoding technique from Lindell and Pinkas [32], which prevents the generator from finding out any information about the evaluator's input if a selective failure attack transpires. CMTB also uses the commitment technique of Kreuter et al. [3] to prevent the generator from swapping the two possible outputs of the oblivious transfer. To ensure the evaluator's input is consistent across all circuits, CMTB uses a technique from Lindell and Pinkas [32], whereby the inputs are derived from a single oblivious transfer.

Generator's input and consistency check The generator sends his input to the cloud for the evaluation circuits. Then the generator, evaluator, and cloud all work together to prove the input consistency of the generator's input. For the generator's input consistency check, CMTB uses the malleable-claw free construction from shelat and Shen [10].

Circuit evaluations The cloud evaluates the garbled circuits marked for evaluation and checks the circuits marked for checking. The cloud enters in the generator and evaluator's input into each garbled circuit and evaluates each circuit. The output for any particular bit is then the majority output between all evaluator circuits. The cloud then recreates each check circuit. The cloud creates the hashes of each garbled

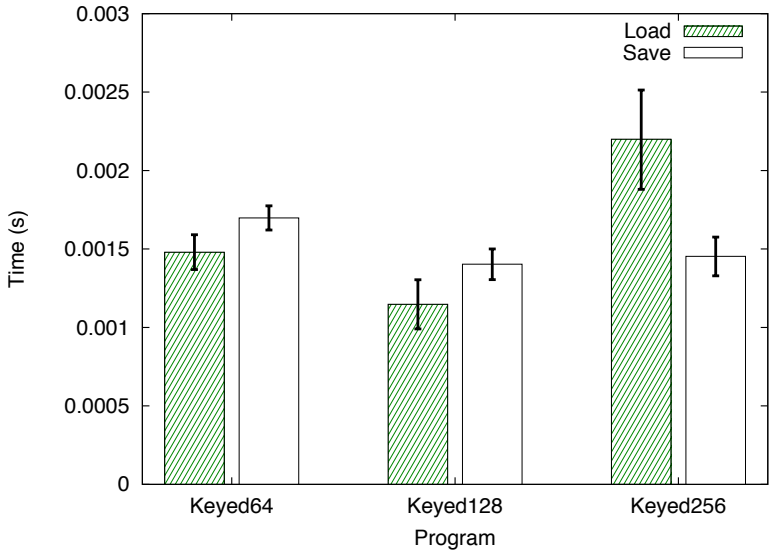


Figure A.1: The amount of time it takes to save and load a bit in PartialGC when using 256 circuits.

circuit and sends those hashes to the evaluator. The evaluator then verifies the hashes are the same as the ones the generator previously committed to.

Output consistency check and output The three parties prove together that the cloud did not modify the output before she sent it to the generator or evaluator. Both the evaluator and generator receive their respective outputs. All outputs are blinded by the respective party's one-time pad inside the garbled circuit to prevent the cloud from learning what any output bit represents.

CMTB uses the XOR one-time pad technique from Kiraz [138] to prevent the evaluator from learning the generator's real output. To prevent output modification, CMTB uses the witness-indistinguishable zero-knowledge proof from Kreuter et al. [3].

A.2 Overhead of Reusing Values

We created several versions of the keyed database program to determine the runtime of saving and loading the database on a per bit basis using our system (See Figure A.1).

This figure shows it is possible to save and load a large amount of saved wire labels in a relatively short time. The time to load a wire label is larger than the time to save a value since saving only involves saving the wire label to a file and loading involves reading from a file and creating the partial input gates. Although not shown in the figure, the time to save or load a single bit also increases with the circuit parameter. This is because we need S copies of that bit - one for every circuit.

A.3 Example Program

In this section we describe the execution of an *attendance application*. Consider a scenario where the host wants each user to sign in from their phones to keep a log of the guests, but also wants to keep this information secret.

This application has three distinct programs. The first program initializes a counter to a number input by the evaluator. The second program, which is used until the last program is called, takes in a name and increments the counter by one. The last program outputs all names and returns the count of users. For this application, users (specifically, their mobile phones) assume the role of evaluators in the protocol (Section 4.3).

First, the host runs the initial program to initialize a database. We cannot execute the second program to add names to the log until this is done, lest we reveal that there is no memory saved (*i.e.*, there is no one else present).

Protocol in Brief: In this first program, the cut-and-choose OT is executed to select the circuit split (the circuits that are for evaluation and generation). Both parties save the decryption keys: the cloud saves the keys attained from the OT and the generator saves both possible keys that could have been selected by the cloud. The evaluator performs the OOT with the other parties to input the initial value into the program. There is no input by the generator so the generator's input check

does not execute. There is no partial input so that phase of the protocol is skipped. The garbled circuit to set the initial value is executed; while there is no output to the generator or evaluator, a partial output is produced: the cloud saves the garbled wire value, which it possesses, and the generator saves both possible wire values (the generator does not know what value the cloud has, and the cloud does not know what the value it has saved actually represents). The cloud also saves the circuit split.

Saved memory after the program execution (when the evaluator inputs 0 as the initial value):

Count
0
Saved Guests

Guest 1 then enters the building and executes the program, entering his name ("Guest 1") as input.

Protocol in Brief for Second Program: In this second program, the cut-and-choose OT is not executed. Instead, both the generator and cloud load the saved decryption key values, hash them, and use those values for the check and evaluation circuit information (instead of attaining new keys through an OT, which would break security). The new keys are saved, and the evaluator then performs the OOT for input. The generator does not have any input in this program so the check for the generator's input is skipped. Since there exists a partial input, the generator loads both possible wire values and creates the partial input gates. The cloud loads the attained values, receives the partial input gates from the generator, and then executes (and checks) the partial input gates to receive the garbled input values. The garbled circuit is then executed and partial output saved as before (although there is more data to save for this program as there is a name present in the database).

After executing the second program the memory is as follows:

Count
1
Saved Guests
Guest1

Guest 2 then enters the dwelling and runs the program. The execution is similar to the previous one (when Guest 1 entered), except that it's executed by Guest 2's phone.

At this point, the memory is as follows:

Count
2
Saved Guests
Guest1
Guest2

Guest 3 then enters the dwelling and executes the program as before. At this point, the memory is as follows:

Count
3
Saved Guests
Guest1
Guest2
Guest3

Finally, the host runs the last program that outputs the count and the guests in the database. In this case the count is 3 and the guests are $Guest1$, $Guest2$, and $Guest3$.

A.4 Proof of the PartialGC Protocol

We formally prove the security of our protocol with the following theorem, which gives security guarantees identical to that of CMTB and the protocol by Kamara et al. [35].

Theorem 12. *The outsourced two-party SFE protocol securely computes a function $f(a, b)$ (as described in Definition 1 for $n = 2$ parties, where a and b are the secret inputs provided by the two parties) in the presence of a third party, the cloud, which knows the public function f, but learns nothing about a and b. We consider the following two corruption scenarios: (1)The Cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, and (2)All but one party is malicious, while the Cloud is semi-honest.*

Proof. To demonstrate that: $\{REAL^{(i)}(k, x; r)\}_{k \in N} \approx \{IDEAL^{(i)}(k, x; r)\}_{k \in N}$, $\forall i \in \{generator, evaluator, cloud\}$, we consider separately each case where a party deviates from the protocol, and then show that these cases reveal no additional information, by considering each point at which the parties interact. □

For the remaining portion of the proof, we shall call the generator, the evaluator, and the cloud, as A, B, and C, respectively. To denote that a party is malicious, we shall use A*, B*, and C*, respectively.

Malicious Evaluator

Both the generator and the cloud participate honestly in the protocol. During the protocol execution, the evaluator only exchanges messages with the other participants at certain points: the cut-and-choose, the oblivious transfer, sending decryption information at the end of the OOT, checking the generator's input consistency, and receiving the proof of validity and output from the garbled circuit. Thus, our simulator need only ensure that these sections of the protocol are indistinguishable to

the adversary - e.g., we need not consider phase 4. Our proof comprises the following hybrid experiments and lemmas:

Simulating the random number generation / coin-flip

$Hybrid1^{(A)}(k, x; r)$: This experiment is the same as $REAL^{(A)}(k, x; r)$ except that instead of running a fair coin toss protocol with A*, the experiment chooses a random string ρ, and a coin-flipping simulator $S_{coinFlip}(\rho, 1^k)$ produces the protocol messages that output ρ.

Lemma 1. $REAL^{(A)}(k, x; r) \approx Hybrid1^{(A)}(k, x; r)$

Proof. Based on the security of the fair coin toss protocol, we know that there exists a simulator $S_{coinFlip}(\cdot, \cdot)$ such that an interaction with $S_{coinFlip}(\cdot, \cdot)$ is indistinguishable from a real protocol interaction. Since everything else in $Hybrid1^{(A)}(k, x; r)$ is exactly the same as in $REAL^{(A)}(k, x; r)$, this proves the lemma. □

Simulating the primitive OT

$Hybrid2^{(A)}(k, x; r)$: This experiment is the same as $Hybrid1^{(A)}(k, x; r)$ except that during the Outsourced Oblivious Transfer, the experiment invokes a simulator S_{OT} to simulate the primitive oblivious transfer operation with A*. The simulator sends A* a random string s and receives the columns of the matrix Q*.

Lemma 2. $Hybrid1^{(A)}(k, x; r) \approx Hybrid2^{(A)}(k, x; r)$

Proof. Based on the malicious security of the OT primitive, we know that there exists a simulator S_{OT} such that an interaction with this simulator is indistinguishable from a real execution of the oblivious transfer protocol. Since everything else in $Hybrid2^{(A)}(k, x; r)$ is identical to $Hybrid1^{(A)}(k, x; r)$, this proves the lemma. □

Checking the output of OOT

$Hybrid3^{(A)}(k, x; r)$: This experiment is the same as $Hybrid2^{(A)}(k, x; r)$ except that the experiment aborts if the matrix Q* is not formed correctly (that is, if A* used inconsistent input values ea^* for any column in generating Q*).

Lemma 3. $Hybrid2^{(A)}(k, x; r) \approx Hybrid3^{(A)}(k, x; r)$

Proof. Let us consider a case in $Hybrid2^{(A)}(k, x; r)$ where for some value of i, A* sends the column value $T^i \oplus ea'$ for some $ea' \neq ea^*$ such that the i^{th} bit is b in ea^* and $b \oplus 1$ in ea'. Then, for every row in Q*, the i^{th} bit will be encrypted in the $b \oplus 1$ entry.

However, when A* sends the value $ea^* \oplus p^*$ to the cloud for decryption, the cloud will decrypt the i^{th} choice, and then it will decrypt the $b \oplus 1$ entry instead of the b entry. This will result in an invalid decryption with probability $1-\epsilon$ for a negligible value of ϵ. This decryption is not (with high probability) a valid commitment key, so the garbled input values won't de-commit, causing the cloud to abort. In $Hybrid3^{(A)}(k, x; r)$, since the experiment observes the messages Q* , p* , and $ea^* \oplus p^*$, it can recover ea^* and check Q* for consistency. (The cloud always aborts if an inconsistency is detected.) □

Simulating consistency check and substituting inputs

$Hybrid4^{(A)}(k, x; r)$: This experiment is the same as $Hybrid3^{(A)}(k, x; r)$ except that the experiment provides a string of $2 \cdot n$ zeros, denoted $0^{2 \cdot n}$, during the consistency check to replace the generator's input.

Lemma 4. $Hybrid3^{(A)}(k, x; r) \approx Hybrid4^{(A)}(k, x; r)$

Proof. Here we cite the proof of shelat and Shen's scheme [4]. Since the messages sent in our scheme are identical to theirs in content, we simply change the entity sending the message in the experiment and the lemma still holds. □

Output and output consistency check

$Hybrid5^{(A)}(k, x; r)$: This experiment is identical to $Hybrid4^{(A)}(k, x; r)$ except that instead of returning the output of the circuit, the experiment provides A* with the result sent from the trusted external oracle.

Lemma 5. $Hybrid4^{(A)}(k, x; r) \approx Hybrid5^{(A)}(k, x; r)$

Proof. Based on the security of garbled circuits, the trusted third party output and the circuit output will be indistinguishable when provided with the input of A*, ea^*. Since we have the k-bit secret key and know the output, we can produce the necessary MAC under the one time pad (similar to Hybrid1). Simulating a majority vote is trivial; the evaluator then decrypts the ciphertext using its OTP key, and then performs a MAC on the output, and compares with the MAC produced within the garbled circuit. This, again, is indistinguishable from the circuit version since we know the output and can ensure that the MACs are identical. □

Malicious Generator

Here, both the evaluator and the cloud participate honestly in the protocol.

Simulating the cut and choose $Hybrid1^{(B)}(k, x; r)$: This experiment is the same as $REAL^{(B)}(k, x; r)$, except during the oblivious transfer (the generator offers up decryption keys for his input as well as for the circuit seed and possible evaluator's input for each circuit), where we randomly choose check and evaluation circuits.

Lemma 6. $REAL^{(B)}(k, x; r) \approx Hybrid1^{(B)}(k, x; r)$

Proof. Since the generator does not learn the circuit split because of the security of the oblivious transfer, this portion of the protocol is indistinguishable from the real version. We can perform all the checks and decryptions required; we also have all the keys needed to check whether we should abort, so a selective failure attack does not help distinguish between the experiment and the real version. □

Simulating the OT $Hybrid2^{(B)}(k, x; r)$: This experiment is the same as $Hybrid1^{(B)}(k, x; r)$ except that rather than run the primitive oblivious transfer with A, the experiment generates a random input string ea' and a random matrix T, then runs a simula-

tor S_{OT} with B*, which gives B* exactly one element from the pair $(T^i, T^i \oplus ea')$ depending on the i^{th} selection bit of B*.

Lemma 7. $Hybrid1^{(B)}(k, x; r) \approx Hybrid2^{(B)}(k, x; r)$

Proof. Based on the security of the primitive OT scheme, we know that the simulator S_{OT} exists, that it can recover B*'s selection bits from the interaction, and that an interaction with it is indistinguishable from a real execution of the OT. Since B* cannot learn any distinguishing information from A's input, again based on the security of the OT primitive, indistinguishability holds between the two experiments. □

Checking the output of OOT $Hybrid3^{(B)}(k, x; r)$: This experiment is the same as $Hybrid2^{(B)}(k, x; r)$ except that the experiment checks the validity of B*'s output from the OOT. Since the experiment possesses T, ea', and s^* (which was recovered by the oblivious transfer simulator S_{OT} in the previous hybrid), the experiment can check whether or not the encrypted set of outputs Y* is well-formed. If not, the experiment aborts.

Lemma 8. $Hybrid2^{(B)}(k, x; r) \approx Hybrid3^{(B)}(k, x; r)$

Proof. Recall that in $Hybrid2^{(B)}(k, x; r)$, if B* does not format the output of the OOT correctly, the cloud will fail to recover a valid commitment key with probability $1 - \epsilon$, where ϵ is negligible in the security parameter. In this case, the committed garbled circuit labels will fail to decrypt properly and the cloud will abort the protocol. In $Hybrid3^{(B)}(k, x; r)$, since the experiment has seen the values Q, s^*, ea', and Y*, it can trivially check to see whether Y* is correctly formed; we about on failure, so a selective failure attack does not help distinguish between the experiment and the real version. Further, B* cannot swap any of A's input labels in the commitments [4]. □

Checking input consistency and recovering inputs

$Hybrid4^{(B)}(k, x; r)$: This experiment is the same as $Hybrid3^{(B)}(k, x; r)$ except that the experiment recovers B*'s input b* during the input consistency check using the random seed recovered in $Hybrid1^{(B)}(k, x; r)$. If the consistency check does not pass or if B*'s input cannot be recovered, the experiment immediately aborts.

Lemma 9. $Hybrid3^{(B)}(k, x; r) \approx Hybrid4^{(B)}(k, x; r)$

Proof. The experiment can perform a hash of the generator's input for each evaluation circuit (we know the split from Hybrid1). If any of these hashes are different, then we know that the generator tried to cheat [4] and can abort if necessary - this is done by the cloud in the real version of the protocol. □

Generating, checking, and evaluating partial input gates

$Hybrid5^{(B)}(k, x; r)$: This experiment is the same as $Hybrid4^{(B)}(k, x; r)$, except for the following: for check circuits, the experiment uses the data sent by the generator, as well as $POut0_{i,j}$, $POut1_{i,j}$, and $CSeed_i$ (recovered during the cut-and-choose) to generate the partial input gates in the same manner as shown in phase 4. It then compares these gates to those the generator sent. If any gate does not match, we know the generator tried to cheat. For evaluation circuits, it finds the point and permute bit and produces the value $GInx_{i,j}$ (this is needed for future hybrids).

Lemma 10. $Hybrid4^{(B)}(k, x; r) \approx Hybrid5^{(B)}(k, x; r)$

Proof. The experiment already has all the $POut$ values from previous hybrids, and receives

$R_i, TT0_{i,j}, TT1_{i,j}$, and bit location ($setPPBitGen$) from B*. It also has $CSeed_i$ from Hybrid1 - thus, it can generate the partial input gates as described in phase 4, and perform all the necessary checks. Since we also know the split from Hybrid1, we can also generate $GInx_{i,j}$ values for the evaluation circuits (which are then entered into the garbled circuit by the cloud in the real version of the protocol). □

Simulating the output check $Hybrid6^{(B)}(k, x; r)$: This experiment is the same as $Hybrid5^{(B)}(k, x; r)$ except that during the output phase the experiment prepares the result received from the trusted third party as the output instead of the output from the circuit.

Lemma 11. $Hybrid5^{(B)}(k, x; r) \approx Hybrid6^{(B)}(k, x; r)$

Proof. Based on the security of garbled circuits, the trusted third party output and the circuit output will be indistinguishable when provided with the input of B*, b^*. Since we have the k-bit secret key and know the output, we can produce the necessary MAC under the one time pad (similar to Hybrid1). Simulating a majority vote is trivial; the experiment then decrypts the ciphertext using its OTP key, and then performs a MAC on the output, and compares with the MAC produced within the garbled circuit. This, again, is indistinguishable from the circuit version to B*, since we know the appropriate OTP key. □

Malicious Cloud

Here, both the generator and the evaluator participate honestly in the protocol.

Replacing inputs for the OOT $Hybrid1^{(C)}(k, x; r)$: This experiment is the same as $REAL^{(C)}(k, x; r)$ except that during the OOT, the experiment replaces A's input ea with a string of zeros $ea' = 0^{2 \cdot l \cdot n}$. This value is then used to select garbled input values from B in the OOT, which are then forwarded to C* according to the protocol.

Lemma 12. $REAL^{(C)}(k, x; r) \approx Hybrid1^{(C)}(k, x; r)$

Proof. In a real execution, C* will observe the random matrix T, the encrypted commitment keys Y, and A's input XOR'd with the permutation string $ea \oplus p$. Since p is random, $ea \oplus p$ is indistinguishable from p. Since T is randomly generated in both $REAL^{(C)}(k, x; r)$ and $Hybrid1^{(C)}(k, x; r)$, they are trivially indistinguishable. Considering the output pairs Y, half of the commitment keys (those not selected

by A) will consist of values computationally indistinguishable from random (since they are XOR'd with a hash), and the keys can only be recovered if C* can find a collision with the hash value $H(j, s)$ without having B's random value s. Thus, C* cannot distinguish an execution of OOT with A's input ea and the simulator's input replacement ea'. Since the rest of the protocol follows $REAL^{(C)}(k, x; r)$ exactly, this proves the lemma. □

Replacing inputs for the consistency check $Hybrid2^{(C)}(k, x; r)$: This experiment is the same as $Hybrid1^{(C)}(k, x; r)$ except that the experiment replaces B's input b with all zeros $0^{2 \cdot n}$. This value is then prepared and checked according to the protocol for consistency across evaluation circuits.

Lemma 13. $Hybrid1^{(C)}(k, x; r) \approx Hybrid2^{(C)}(k, x; r)$

Proof. In this hybrid, C* observes a set of garbled input wire values from B. Based on the security of garbled circuits, observing one set of garbled input wire values is indistinguishable from observing any other set of input wire values, such that C* cannot distinguish between the garbled input for b and the garbled input for $0^{2 \cdot n}$. The rest of the hybrid is the same as $Hybrid1^{(C)}(k, x; r)$. □

Partial gates $Hybrid3^{(C)}(k, x; r)$: This experiment is the same as $Hybrid2^{(C)}(k, x; r)$ except that the experiment sends the appropriate truth table information, R_i, $POut$ values, and the bit location from $setPPBitGen$ to C*.

Lemma 14. $Hybrid2^{(C)}(k, x; r) \approx Hybrid3^{(C)}(k, x; r)$

Proof. In this hybrid, C* observes a set of garbled wire and truth table values from B. Since these partial gates are identical to garbled circuit gates as far as operation is concerned, the security of garbled circuits ensures that observing one set of garbled input wire values is indistinguishable from observing any other set of wire values, such that C* cannot distinguish between the garbled value for b and the garbled value for $0^{2 \cdot n}$. The rest of the hybrid is the same as $Hybrid2^{(C)}(k, x; r)$. □

Checking the output

$Hybrid4^{(C)}(k, x; r)$: This experiment is the same as $Hybrid3^{(C)}(k, x; r)$ except that after the circuit is evaluated, the experiment checks that the result output by C* is as expected; otherwise, the experiment immediately aborts.

Lemma 15. $Hybrid3^{(C)}(k, x; r) \approx Hybrid4^{(C)}(k, x; r)$

Proof. After the cloud sends the output (under the OTP from phase 1) to us, we can decrypt, since we have the OTP from previous hybrids. We then perform a MAC and compare with the MAC calculated within the garbled circuit to verify that C* did not modify the output. □

Appendix B

Frigate

B.1 Example Programs

Input Example: The program below gives an example of much of the syntax in Frigate's input language, using statements, types, and input and output. Each output is the addition of the two inputs.

```
#define wiresize 32
#parties 2
typedef int_t wiresize int
typedef struct_t mystruct {
    int x;
}
typedef struct_t newstruct {
    int x;
    newstruct var[5];
}
#input 1 int
#output 1 int
#input 2 int
#output 2 int
function void main(){
    output1 = input1 + input2;
    output2 = input1 + input2;
}
```

	Time(s)	All Gates	Non-XOR	Time(s)	All Gates	Non-XOR
Program	Frigate			PCF		
Hamming 1000	0.0067 ± 7%	17,829	4,691	5.1 ± 1%	21,970	4,882
Hamming 16384	0.053 ± 4%	294,737	77,273	6.95 ± 0.8%	391,683	96,117
Mult 256	0.038 ± 5%	391,171	131,330	54.2 ± 0.7%	1,659,808	400,210
Mult 4096	7.94 ± 0.5%	100,630,531	33,558,530	63.7 ± 0.9%	364,605,460	89,444,609
Matrix Mult 5	0.011 ± 2%	372,377	128,252	60.2 ± 0.4%	433,475	127,225
Matrix Mult 16	0.17 ± 1%	12,201,986	4,202,498	68.8 ± 0.4%	14,308,864	4,186,368
AES	0.009 ± 10%	34,889	10,383	0.48 ± 5%	38,260	12,578
RSA 256	0.306 ± 0.5%	942,210,819	202,441,218	272 ± 0.6%	673,105,990	235,925,023
RSA 512	1.08 ± 0.2%	7,526,940,163	1,615,070,210	275 ± 0.8%	5,397,821,470	1,916,813,808
Program	CBMC			KSS		
Hamming 1000	0.71 ± 2%	54,233	18,906	2.2 ± 1%	20,493	4,641
Hamming 16384	1.16 ± 0.8%	910,495	290,728	4.21 ± 0.8%	370,110	88,952
Mult 32	0.48 ± 1%	6,223	1,741	0.34 ± 6%	15,935	5,983
Mult 256	9,800* ± 5%	5,880,833	2,264,860	17 ± 2%	1,044,991	391,935
Matrix Mult 5	1.8 ± 2%	795,988	223,720	32 ± 2%	1,968,452	746,177
Matrix Mult 16	1,500* ± 7%	26,182,494	7,251,991	1900 ± 5%	64,570,969	24,502,530
AES	0.60 ± 4%	35,607	11,469	0.71 ± 1%	49,912	15,300
RSA 256	-**	-	-	14,000 ± 4%*	928,671,864	315,557,288
Program	Obliv-C (time and # of non-XOR gates)			ObliVM (time and # of non-XOR gates)		
Hamming 1000	0.916 ±4%	7,719		0.148 ±7%	1,989	
Hamming 16384	0.962 ±6%	126,945		0.150 ±5%	32,752	
Mult 256	0.978 ±3%	95,296		0.837 ±3%	523,776	
Mult 4096	0.927 ±4%	146,292,736		0.844 ±4%	134,209,536	
Matrix Mult 5	0.942 ±5%	127,969		0.851 ±5%	653,125	
Matrix Mult 16	0.919 ±5%	4,194,273		0.862 ±5%	139,591,680	
AES	1.549 ±4%	385,056		0.198 ±8%	61,227	
RSA 256	1.031 ±5%	169,993,375		-	-	
RSA 512	1.083 ±6%	3,525,298,575		-	-	

Table B.1: This table shows the compile time in seconds, the amount of total gates, and the amount of non-XOR gates. Note that for CBMC and KSS, we ran Mult 32 and Mult 256 instead of Mult 256 and Mult 4096.
All tests were ran 10 times unless otherwise noted: * tests ran 3 times, ** we stopped compilation after 6 hours.

Procedure Example: This is the example program discussed in Section 5.6.4.

```
function void main(){
    int x = input1;
    int y = input2;
    for(int i=0;i<1000;i++) {
        x = x + y + y + y + y + y;
    }
    output1 = x;
}
```

Table B.1 gives compiler performance results for our different test programs.

B.2 Frigate Design Details

This section lists some of our design decisions and explains why we decided to do things differently from other compilers.

No Recursion: We do not allow recursion in our execution model. Multiple copies of a specific function could be created to simulate recursion but this is not done as part of a native operation. Since the depth of recursion must be known at compile time, this does not remove functionality from the language but may reduce expressiveness.

void Functions: Our compiler allows functions that return nothing. At first this may seem counter intuitive to our model since nothing performed in these functions would ever be used. However, extensions like adding the ability to pass variables to functions by "reference" may require the use of *void* functions. We therefore included them into our compiler.

Return Must be the Last Statement In a Function: In an effort to reduce complexity of the compiler, we require any return statements to be at the end of a function. *void* functions do not need a return statement but may have a return statement as the last statement in the function.

Typed Constants: Constants can be typed to a specific length and sign. By default, constants are not typed, sized to their bit-length + 1, and use unsigned operators when all operands are untyped constants. In the presence of typed operands and untyped constants, the operator will use the sign of the typed operand. We added the # and ## operators to specify the bit-length and type of constants. We use # to represent a signed constant and ## to represent an unsigned constant, *i.e.*, `char16 x = -9#16;` defines -(9) to be signed 16-bit number. We issue a warning if one of these operators is not used with a constant, as it may produce incorrect results with negative numbers.

Extending and Reducing Operators: In order to reduce the size of some circuits, we provide a multiplication operator (∗∗) that takes in two X-bit numbers

and produces a $X * 2$-bit number, and operators for division (//) and modulus (%%) that take in a dividend of size X, quotient of size $ceiling((X + 1)/2)$, and returns a $ceiling((X + 1)/2)$-bit result. We note the modulus and division operator only work correctly when the result can completely fit in the quotient. These circuits are much smaller than in the case of using X to X-bit operators when, in the case of multiplication, a $X * 2$-bit result is desired. *Note:* We do not allow the use of untyped constants in these operators.

Rotation on Constants: We allow constants to be rotated but *beware*, constants are sized to their bit-length + 1 by default. If a user rotates a constant and that constant did not have a defined bit-length, strange results may occur.

Array Size and Wire Amount Declarations Cannot use Variables: Our compiler was made with fairly strict type checking. The sizes of arrays and the amount of wires selected (with the wire operand) must be known during the type-checking phase. As type checking is after #define replacement, terms replaced by constant expressions from #define in these declarations is accepted by our compiler.

Constants Up to 64 Bits in Length: Constants can be defined to a length of up to 64 bits in base 10. This is quite simple to extend in the future as the constant values themselves are represented as strings.

More Than Two Parties: Our compiler allows more than two parties in the computation unlike the other compilers we examined. Adding additional parties to the computations can be useful to declare different types of output.

Although standard example programs do not use more than two parties, work such as PartialGC [20] requires three input and output parties even though only two parties have input and output in the secure computation(the third party‘s input and output are saved wire values).

B.3 Frigate Validation Details

To practically validate a compiler, we must check all possible ways that each statement can output a sub-circuit and the ways in which data can flow from the beginning to the end of the program (*e.g.*, when the data is encapsulated in variables, when it is used in control structures, etc.). While daunting, the task is made simpler by realizing that each operator and control structure can only be output in a finite number of ways, *i.e.*, an *if/else* statement has 2 possibilities: it is either the first *if/else* statement or is nested under at least one other *if/else* statement.

We perform the tests outlined in Section 5.3. At the conclusion of these tests we have covered most, if not all, the state space in Frigate. We have tested (1) the correctness of each mini-circuit an operator uses, (2) all the ways in which data can populate each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, and arrays declarations), (4) common edge cases, and (5) unique constructs to Frigate's input format. Some tests, like verifying whether a file is correctly included, were performed by printing out the AST and not by compiling and executing the program.

For each type of test, we test a variety of positive (correct) and negative (incorrect) results with emphasis on edge cases. For instance, for the addition operator we test the following (1) Does the operator correctly perform with all possible unsigned 8-bit input combinations? (2) Does the operator correctly perform with all possible signed 8-bit input combinations? (3) Does adding two different types of the same length give a warning? (4) Does adding two different types of different lengths give an error?

Operators: The first tests on operators examine whether the sub-circuits, or templates, for each operator (addition, subtraction, etc.) are correct. For each of these operations, we constructed a large program to test all possible input combinations of 8-bits. This involves 65,536 tests for each binary operator for both signed and unsigned values. An short example of these tests is seen in Figure B.1.

```
...
output1{test++}=1##1==(68##8<107##8);
output1{test++}=1##1==(68##8<=107##8);
output1{test++}=0##1==(68##8>107##8);
output1{test++}=0##1==(68##8>=107##8);
...
```

Figure B.1: A subset of the testing performed on Frigate. In this specific program, output1 is an output with 393,216 bits. Each bit is set to whether a specific output value was correct. We then automatically the output for all 1s.

```
if(x) { if(y) { } else { } }
else { if(z) { } else { } }
```

Figure B.2: Twice nested *if* statements. There are 8 possible combinations as x, y, and z can either be 0 or 1.

Once we know the template circuits are correct, we must then show all possible types of data that can be entered into the template work as well. These types are: (1) constants, (2) variables, or (3) results from an expression. Once these tests pass, we know the operator can correctly input the different possible types of data.

Control Structures: Once each operator is shown to be correct, we know any other errors found will not be from the primitive operators but from the control structures. There are four control structures in Frigate we must test: functions, *if/else* statements, *for* loops, and procedures.

For each control structure, we check every way in which it can be output. Conditional *if/else* statements can be checked for correctness by performing an exhaustive search up to depth 2 (*i.e.*, test all 8 possible cases of the conditional output as shown in Figure B.2). The unique ways to output the *if/else* statement occur within the first conditional, while the depth-2 *if/else* statement must be combined with its parent conditional. If the nested *if/else* conditionals combine correctly at depth 2 then by induction, it will work for subsequently nested conditionals as well. Each *if/else* statement should be tested for when the guard values are dependent on user input as well as when they are not dependent on user input.

It is relatively simple to validate the correctness of *for* loops when they are only nested under another *for* loop by checking whether they output the circuit the correct number of times. When they are used under *if/else* statements, problems can arise depending on how the loop variable is scoped and whether the loop variable's result will be labeled *UNKNOWN*, meaning the result is based on user input due to the *if* statement, or whether it will be labeled as a 0 or 1 value. We test to depth 2 in case there is an external state used by the compiler that may prevent nested *for* loops from working correctly under *if/else* statements.

Functions also have a finite number of possible states to test. Our procedure for carrying out this testing was as follows. (1) Test the function call operator, where a function call is treated as any other operator that takes in any number of operands (parameters) and returns a single operand. We test different possible combinations of parameters up to length 2, as that is where data no longer acts in a unique way. (2) Function definitions need to be tested to ensure different types of return variables (array, struct, int, uint) work correctly. (3) Test two of the same function call in an operand with different results (*e.g.*, `addX(3,4) + addX(5,6)`). It is possible that the results of the first call may be overwritten by the second call. (4) Test that parameters can be used inside of the functions.

Although we do not have global variables in our input language, we suggest the following two tests for compilers that use global variables. (1) Test whether functions correctly modify global variables. (2) Test when functions are called under an *if/else* statement and whether the function modifies the global variable as expected, *i.e.*, if the guard is *false* then the global variable is not modified.

The correctness of procedures reduces to a simple question: *is each variable composed of the exact same wires every iteration?* We know that variables will use the same free wires if the wire pool is sorted before each iteration. Procedures can be difficult to use and if procedures are not used correctly by a developer, then a program

will output undesired results. However this is not a *correctness* problem.

We test various specific cases: Empty functions, array declarations and access up to 2 dimensions (depth 2 as each dimension is compartmentalized, if it works for depth 1 and depth 2, then it works for the base case and when 'nested'), verify that *#includes* and *#defines* work correctly, check that the *parties* variable correctly determines what input and output variables can be used, assignments of arrays and structs, as well as the additional edge cases mentioned above.

Frigate's Interpreter: In order to use the circuits generated by Frigate, we also validate the correctness of the interpreter. If the interpreter was created first in order to test the functionality of the compiler then it will correctly function after all the functionality of the compiler was checked (*i.e.*, there are no more edge cases to check where it would fail). If the interpreter was not created first then each test should be run through the interpreter.

In an effort to help others build an interpreter like ours we present the following errors we found after we had our interpreter running: (1) where the last operation in each function was not called, (2) where the last operation in each function was performed twice, and (3) calling empty functions did not work correctly.

Complete Validation Set: We performed a much more extensive suite of tests on Frigate than we performed on the other compilers. We went looking for where we believed we would find errors in Frigate and in some cases we found errors during validation. Other than the millions of arithmetic tests we performed, our validation test set has more than 17,000 tests. This set contains the above test cases and has many arithmetic tests. Most of the tests were generated using various Java programs, *i.e.*, we could template the problem of nested if statements and then generate all possible combinations.

Once we created our set of validation set program, we integrated it to be performed every time `make` is called. This way, we can see if any changes we made to the compiler

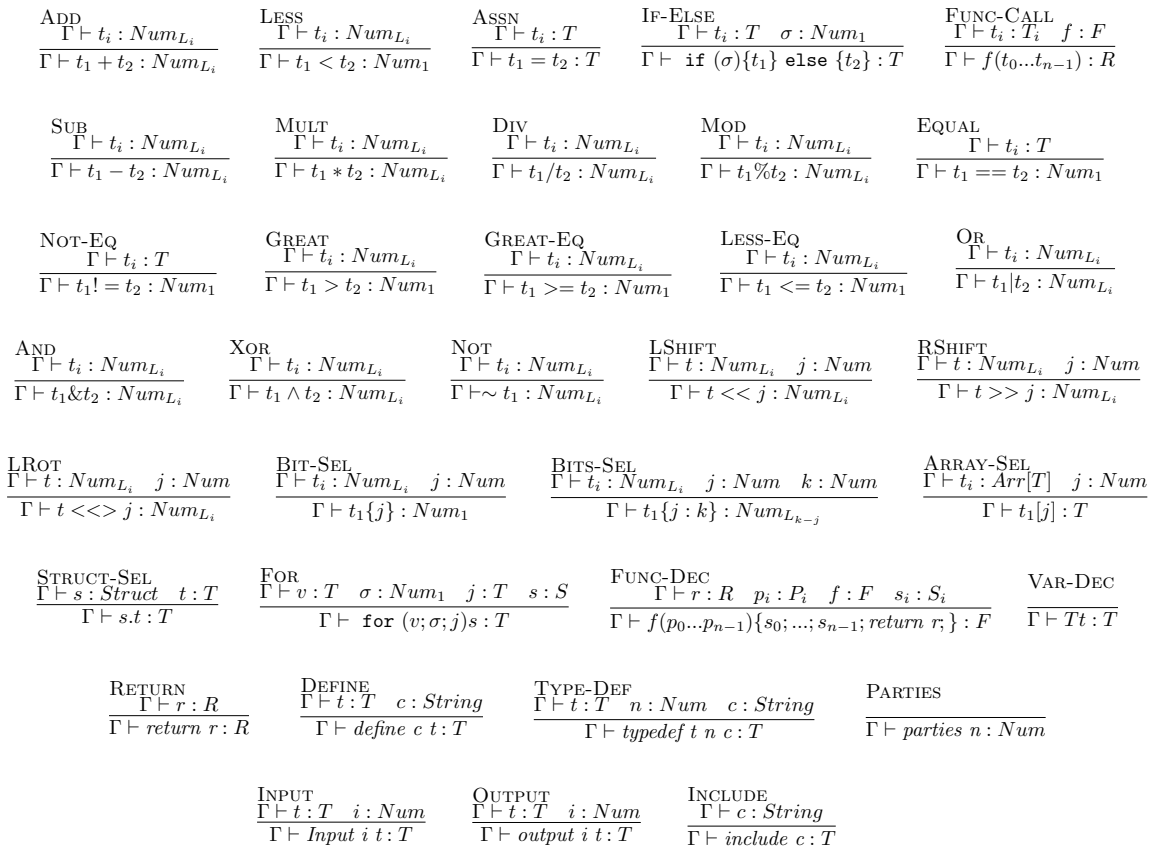


Figure B.3: The full set of typing rules for all Frigate operators.

broke a test case (and this has been useful).

B.4 Operator Typing

In Figure B.3 we give the full typing rules for our language. Each operator or statement has its own typing rule.

Each of the typing rules has a set of types on the top i.e., $t : T$ means t is of type T in the program context Γ. The bottom is a statement in Frigate's input, along with the types of each item. Num is a number of any bit-length. Num_{L_i} is a number of bit-length i. R, T, F, S, P can be replaced with any type but use different names to differentiate where they are used.

Bibliography

[1] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'82)*, 1982.

[2] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the USENIX Security Symposium (SECURITY '11)*, 2011.

[3] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium (SECURITY'12)*, 2012.

[4] abhi shelat and Chih-Hao Shen. Fast Two-Party Secure Computation with Minimal Assumptions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'13)*, 2013.

[5] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay–A Secure Two-Party Computation System. In *Proceedings of the USENIX Security Symposium (SECURITY'04)*, 2004.

[6] Benjamin Mood, Lara Letaw, and Kevin Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.

[7] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-party Computations in ANSI C. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[8] Benjamin Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of the USENIX Security Symposium (SECURITY '13)*, 2013.

[9] DARPA. DARPA "Brandeis" program aims to ensure online privacy through technology. `http://www.darpa.mil/NewsEvents/Releases/2015/03/11.aspx`, 2015.

[10] abhi shelat and Chih-Hao Shen. Two-Output Secure Computation with Malicious Adversaries. In *Proceedings of Advances in Cryptology (EUROCRYPT'11)*, 2011.

[11] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the USENIX Conference on Security (SECURITY'10)*, 2010.

[12] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC '09)*, 2009.

[13] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for Automating Secure Two-Party Computations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[14] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *Proceedings of the ACM Conference on Computer Communications Security (CCS'13)*, 2013.

[15] 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. Secure multiparty computation goes live. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'09)*, 2009.

[16] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets'12)*. ACM, 2012.

[17] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multiparty computation for financial data analysis. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.

[18] Giovanni Di Crescenzo, Joan Feigenbaum, Debayan Gupta, Euthimios Panagos, Jason Perry, and Rebecca N. Wright. Practical and privacy-preserving policy compliance for outsourced data. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'14)*, 2014.

[19] Martin Libicki, Olesya Tkacheva, Chaoling Feng, and Brett Hemenway. *Ramifications of DARPA's PROCEED Program.* RAND, Santa Monica, 2014.

[20] Benjamin Mood, Debayan Gupta, Kevin Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'14)*, 2014.

[21] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. *Proceedings of the IEEE European Symposium on Security and Privacy (Euro S&P'16)*, 2016.

[22] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'16)*, 2016.

[23] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N. Wright. Systematizing secure computation for research and decision support. In *Journal of Security and Communication Networks (SCN'14)*. 2014.

[24] Mihir Bellare and Silvio Micali. Non-Interactive Oblivious Transfer and Applications. In *Proceedings of Advances in Cryptology (CRYPTO'89)*, 1990.

[25] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *Proceedings of Advances in Cryptology (CRYPTO'03)*, 2003.

[26] Moni Naor and Benny Pinkas. Oblivious Transfer and Polynomial Evaluation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'99)*, 1999.

[27] Moni Naor and Benny Pinkas. Efficient Oblivious Transfer Protocols. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, 2001.

[28] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the ACM conference on Computer and Communications Security (CCS'08)*, 2008.

[29] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'02)*, 2002.

[30] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *Proceedings of Advances in Cryptology (EUROCRYPT'08)*, 2008.

[31] Mehmet S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[32] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Proceedings of Advances in Cryptology (EUROCRYPT'07)*, 2007.

[33] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the Conference on Theory of Cryptography (TCC'11)*, 2011.

[34] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[35] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: A System for Server-Aided Secure Function Evaluation. In *Proceedings of the ACM conference on Computer and Communications Security (CCS'12)*, 2012.

[36] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium (SECURITY'13)*, 2013.

[37] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the Computer Security Applications Conference*, 2014.

[38] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Outsourcing Secure Two-Party Computation as a Black Box. In *Proceedings of the International Conference on Cryptology and Network Security (CANS)*, 2015.

[39] Henry Carter, Chaitrali Amrutkar, Italo Dacosta, and Patrick Traynor. For Your Phone Only: Custom Protocols For Efficient Secure Function Evaluation On Mobile Devices. *Journal of Security and Communication Networks (SCN'14)*, 2014.

[40] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'08)*, 2008.

[41] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure Two-Party Computation is Practical. In *Proceedings of Advances in Cryptology (ASIACRYPT'09)*, 2009.

[42] Florian Kerschbaum. Expression Rewriting for Optimizing Secure Computation. In *Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013.

[43] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-Homomorphic Encryption and Multiparty Computation. In *Proceedings of Advances in Cryptology (EUROCRYPT'11)*, 2011.

[44] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic Evaluation of the AES Circuit. In *Proceedings of Advances in Cryptology (CRYPTO'12)*, 2012.

[45] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*, 2008.

[46] Louis Kruger, Somesh Jha, Eu-Jin Goh, and Dan Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'06)*, 2006.

[47] Seny Kamara, Payman Mohassel, Mariana Raykova, and Saeed Sadeghian. Scaling Private Set Intersection to Billion-Element Sets. Technical Report MSR-TR-2013-63, Microsoft Research, 2013.

[48] 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. Secure Multiparty Computation Goes Live. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'09)*, 2009.

[49] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian Tax and Customs Board evaluated a tax fraud detection system based on secure multi-party computation. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'15)*, 2015.

[50] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'13)*, 2013.

[51] Craig Gentry, Sergey Gorbunov, Shai Halevi, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. How to Compress (Reusable) Garbled Circuits. Cryptology ePrint Archive, Report 2013/687, 2013. `http://eprint.iacr.org/`.

[52] Luís T. A. N. Brandão. Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique. Technical report, University of Lisbon, 2013.

[53] Steve Lu and Rafail Ostrovsky. How to Garble RAM Programs. In *Proceedings of Advances in Cryptology (EUROCRYPT'13)*. 2013.

[54] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM Revisited. In *Proceedings of Advances in Cryptology (EUROCRYPT'14)*, 2014.

[55] Steve Lu and Rafail Ostrovsky. Garbled RAM Revisited, Part II. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

[56] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure Computation on the Web: Computing Without Simultaneous Interaction. In *Proceedings of Advances in Cryptology (CRYPTO'11)*, 2011.

[57] Yan Huang, Jonathan Katz, and David Evans. Quid-Pro-Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. *Proceedings of the IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[58] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS'12)*, 2012.

[59] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of Advances in Cryptology (CRYPTO'12)*, 2012.

[60] Thomas Schneider and Michael Zohner. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'13)*, 2013.

[61] Tony Hoare. The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*, pages 25–35. Springer, 2003.

[62] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[63] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'11)*, 2011.

[64] Lopes, Nuno P. and Monteiro, José. Weakest Precondition Synthesis for Compiler Optimizations. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

[65] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. A Witnessing Compiler: A Proof of Concept. In *Proceedings of the Conference on Runtime Verification*, 2013.

[66] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, 1998.

[67] George Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[68] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'11)*, 2011.

[69] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. Testing-Based Compiler Validation for Synchronous Languages. In *NASA Formal Methods*, pages 246–251. Springer, 2014.

[70] Cheng Wang, Rengan Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez. A Validation Testsuite for OpenACC 1.0. In *IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW'14)*, 2014.

[71] Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman. An openmp 3.1 validation testsuite. In *OpenMP in a Heterogeneous World*, pages 237–249. Springer, 2012.

[72] SuperTest Compiler Test and Validation Suite. `http://www.ace.nl/compiler/supertest.html`.

[73] Arm Compiler Verification Process. `http://www.arm.com/products/tools/software-tools/mdk-arm/compilation-tools/compiler-verification.php`.

[74] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. `http://oblivc.org/category/paper.html`, 2015.

[75] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[76] Alred Aho, Monica Lam, Ravi Sethi, and Jeffery Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[77] Plum Hall, Inc. The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`.

[78] flex: The Fast Lexical Analyzer. `http://flex.sourceforge.net`.

[79] Gnu bison. `http://www.gnu.org/software/bison/`.

[80] Joan Boyar, René Peralta, and Denis Pochuev. On the multiplicative complexity of boolean functions over the basis $(\wedge, \oplus, 1)$. *Theoretical Computer Science*, 235(1):43 – 57, 2000.

[81] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[82] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits using Half Gates. In *Proceedings of Advances in Cryptology (EUROCRYPT'15)*, 2015.

[83] Bellare, Mihir and Hoang, Viet Tung and Keelveedhi, Sriram and Rogaway, Phillip. Efficient Garbling from a Fixed-Key Blockcipher. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[84] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "Free-XOR" technique. In *Proceedings of the Conference on Theory of Cryptography (TCC'12)*, 2012.

[85] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR. In *Proceedings of Advances in Cryptology (CRYPTO'14)*, 2014.

[86] Ayush Rastogi, Matthew Hammer, Michael Hicks, et al. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[87] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[88] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[89] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[90] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[91] Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. Efficient implementation of rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES'03)*, 2003.

[92] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[93] V Vipindeep and Pankaj Jalote. List of common bugs and programming practices to avoid them. *Electronic*, March 2005.

[94] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX'00)*, 2000.

[95] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*, 2012.

[96] Úlfar Erlingsson and Martín Abadi. Operating system protection against side-channel attacks that exploit memory latency. Technical report, Microsoft Research, 2007.

[97] Dag Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of Topics in Cryptology (CT-RSA'06)*, 2006.

[98] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Conference on Computer Communications Security (CCS'13)*, 2013.

[99] Niels Ferguson. AES-CBC+ Elephant diffuser: A disk encryption algorithm for Windows Vista. Technical report, Microsoft, 2006.

[100] Microsoft-TechNet. Bitlocker drive encryption overview. `https://technet.microsoft.com/en-us/library/cc732774.aspx`, 2008.

[101] Trusted Computing Group. Trusted platform module main specification (tpm1.0). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2011.

[102] Trusted Computing Group. Trusted platform module library specification (tpm2.0). `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`, 2013.

[103] David Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333. Springer, 1996.

[104] Sumit Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2014.

[105] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *Proceedings of the Conference on Innovative Data Systems Research*, 2013.

[106] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[107] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the ACM Conference on Computer Communications Security (CCS'13)*, 2013.

[108] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the International Conference on Supercomputing*, 2003.

[109] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.

[110] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. *ACM SIGOPS Operating Systems Review*, 42:315–328, 2008.

[111] Masaji Kawahara. Superdistribution: the concept and the architecture. *Transactions of the Institute of Electronics, Information and Communication Engineers (1976-1990)*, 73(7):1133–1146, 1990.

[112] Sean Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.

[113] James Greene. Intel trusted execution technology. *Intel Technology White Paper*, 2012.

[114] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*, 2008.

[115] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News*, 42:67–80, 2014.

[116] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[117] Ronald Cramer and Ivan Damgård. *Multiparty computation, an introduction*. Springer, 2005.

[118] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols – Techniques and Constructions*. Springer, 2010.

[119] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing: An Information Theoretic Approach*. Self-published manuscript, 2013. `https://users-cs.au.dk/jbn/mpc-book.pdf`.

[120] Joan Feigenbaum, Benny Pinkas, Raphael Ryger, and Felipe Saint-Jean. Secure computation of surveys. In *EU Workshop on Secure Multiparty Protocols*, 2004.

[121] Dan Bogdanov, Liina Kamm, Sven Laur, and Pille Pruulmann-Vengerfeldt. Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826, 2013. `http://eprint.iacr.org/2013/826`.

[122] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N. Wright. The secure computation annotated bibliography, 2014. `http://paul.rutgers.edu/~jasperry/ssc-annbib.pdf`.

[123] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Proceedings of the Conference on Theory of Cryptography (TCC'07)*, 2007.

[124] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'04)*, 2004.

[125] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'93)*, 1993.

[126] Shafi Goldwasser and Yehuda Lindell. Secure computation without agreement. In *Proceedings of the 16th Int'l Symposium on Distributed Computing*, 2002.

[127] Martin Hirt, Christoph Lucas, Ueli Maurer, and Dominik Raub. Graceful degradation in multi-party computation (extended abstract). In *Proceedings of the Conference on Information Theoretic Security*, 2011.

[128] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In *Proceedings of the Conference on Theory of Cryptography (TCC'12)*, 2012.

[129] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. Cryptology ePrint Archive, Report 2014/075, 2014. `http://eprint.iacr.org/`.

[130] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'08)*, 2008.

[131] Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the ACM Conference on Computer Communications Security (CCS'12)*, 2012.

[132] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. On 2-round secure multiparty computation. In *Proceedings of Advances in Cryptology (CRYPTO'02)*, 2002.

[133] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'88)*, 1988.

[134] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *Proceedings of the ACM Symposium on Theory of Computing (STOC'89)*, 1989.

[135] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *Proceedings of the ACM Symposium on Theory of Computing (STOC'86)*, 1986.

[136] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'87)*, 1987.

[137] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In *Proceedings of Advances in Cryptology (EUROCRYPT'03)*, 2003.

[138] Mehmet S. Kiraz and Berry Schoenmakers. A Protocol Issue for the Malicious Case of Yao's Garbled Circuit Construction. In *Proceedings of Symposium on Information Theory in the Benelux*, 2006.