Techniques for Scalable Secure Computation Systems

Benjamin Kreuter

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy in the Computer Science Department Northeastern University

Supervisor: Prof. abhi shelat

April 2018
THESIS TITLE: Techniques for Scalable Secure Computation Systems

AUTHOR: Benjamin Kreuter

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor

Date

Thesis Reader

Date

Thesis Reader

Date

Thesis Reader

Date

Thesis Reader

Date

GRADUATE SCHOOL APPROVAL:

Director, Graduate School

Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature

Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
# Contents

1 Introduction 9  
1.1 Contributions of this Thesis ................................. 10  
1.2 Related Work .............................................. 11  
1.2.1 MPC Protocols ........................................ 11  
1.2.2 MPC Frameworks ........................................ 13  
1.2.3 Real-World MPC Applications ........................... 14  

2 Background 17  
2.1 Security Model ............................................ 17  
2.1.1 Formal Security Model ................................ 18  
2.1.2 The Random Oracle Model .............................. 18  
2.2 Computation with Circuits ................................ 19  
2.2.1 Garbled Circuits ...................................... 19  
2.2.2 Quadratic Arithmetic Programs ......................... 20  
2.3 Cryptographic Primitives .................................. 20  
2.3.1 Diffie-Hellman Protocols ............................... 20  
2.3.2 Pairing-Friendly Elliptic Curves ....................... 22  
2.3.3 Homomorphic Encryption ............................... 22  

I Generic Protocols 25  

3 Billion-Gate Secure Computation with Malicious Adversaries 27  
3.1 Introduction ............................................... 27  
3.2 Security Techniques ....................................... 31  
3.3 Performance Techniques ................................... 33  
3.3.1 Baseline Yao’s Garbled Circuit ....................... 34  
3.3.2 Free-XOR ............................................... 34  
3.3.3 Garbled Row Reduction ............................... 35  
3.3.4 Random Seed Checking ............................... 35  
3.3.5 Working with Large Circuits ......................... 36  
3.4 The Full Protocol ......................................... 37  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6.3</td>
<td>Prototype Performance</td>
<td>124</td>
</tr>
<tr>
<td>7.7</td>
<td>Chapter Acknowledgements</td>
<td>126</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>129</td>
</tr>
</tbody>
</table>
Abstract

Secure Multiparty Computation protocols allow a set of mutually distrustful parties to compute a function on their respective inputs while guaranteeing both the privacy of the inputs and the correctness of the outputs. Despite thirty years of research and many potential real-world applications, MPC deployment remains limited by efficiency and scalability bottlenecks. This thesis presents novel protocol designs and engineering techniques to address those performance and scalability issues.

In Part I three generic systems that can be applied to arbitrary functionalities are presented: two systems for garbled circuits protocols, and one in the setting of verifiable computation with succinct non-interactive arguments of knowledge (SNARKs). Both the protocols and the larger runtime systems are carefully designed to scale to arbitrary computations, and use techniques from compilers to automatically optimize the circuits. In the setting of garbled circuits, a protocol is presented with security against malicious adversaries that is carefully designed to avoid scalability bottlenecks. Runtime systems using that garbled circuit protocol that support arbitrarily large circuits are described, and experimental results are presented. In the SNARK setting, a construction is proposed that balances scalability with practicality, using a limited form of proof bootstrapping.

In Part II two custom protocols are presented, which support applications that have been deployed as a large technology company. The constraints imposed by those applications require communication to be minimized, even at the cost of higher overall runtime, and rule out solutions based on generic protocols. The first protocol computes a generalization of private set intersection called Private Intersection-Sum, using a variation of the well-known Diffie-Hellman based PSI protocol. The second protocol computes an average of vectors from a large number of mobile phones, requiring a careful construction to handle the high failure rate of such devices.
Secure multiparty computation protocols (MPC) allow a group of mutually distrustful parties to compute a function over their respective inputs without revealing those inputs while guaranteeing the privacy and correctness of their respective outputs. MPC has attracted significant research attention for several decades among theoreticians and recently there have been efforts to create practical MPC systems. Early research on MPC established the feasibility of generic MPC protocols [Yao82a, GMW87]. Researchers have also presented custom MPC protocols for specific functionalities. More recently, researchers have developed end-to-end frameworks for generic protocols, which improve usability by allowing functionalities to be expressed in a high-level language.

Theoretical research has focused on establishing security definitions that capture the security goals of MPC, and proving that for any functionality these definitions can be satisfied. At a high level, the security goal of MPC is to emulate a trusted third party using a cryptographic protocol. By showing that any attack against the real protocol is equivalent to some attack that only interacts with such a trusted party a protocol can be proved secure. Informally, for any coalition of attacking parties, a proof of security must show that the output distributions of all parties in the real protocol is indistinguishable from the distributions of the corresponding “ideal world” parties. Extensions of this definition cover various cases of sequential [Can98, BCF+06] and concurrent composition [Can01, CLOS02, CCL15], as well as weaker security notions [BS05a, AL10, HKE12, GJ15].

Beyond their theoretical value, generic protocols also promise greatly reduced engineering effort for MPC applications. Designing a custom secure computation protocol for some functionality and proving its security can be very labor-intensive. Worse, extending a custom protocol to support a seemingly simple extension of the functionality may not be so simple, possibly requiring an entirely new design and security proof. With a generic protocol, designing an MPC application could be reduced to stating the functionality at a higher level; security follows from the security of the generic protocol, without requiring a separate security proof for each functionality.

Unfortunately, despite the real-world need for MPC, practical applications have remained limited. Although researchers have continued to improve end-to-end running times, significant challenges remain. Generic protocols typically impose significant resource costs, especially in terms of the amount of communications required. On the other hand, while custom protocols are often more resource-efficient, they are difficult to design for arbitrarily complex functionalities and require more engineering work to implement and use.
1.1 Contributions of this Thesis

This thesis studies systems level challenges in implementing generic MPC protocols, as well as custom protocols for application-specific tasks that outperform generic solutions.

Chapter 3 presents BillionYao, a 2-party generic protocol with security against malicious adversaries and a carefully engineered compiler and runtime system, based on the work of Kreuter, shelat, and Shen [KSS12]. BillionYao was the first complete system supporting circuits with billions of gates in the malicious model. The BillionYao garbled circuits protocol is similar to the protocol presented by shelat and Shen [SS11], but uses Lindell and Pinkas’ random input replacement technique [LP11a] and a modification of Kiraz’s approach to output authenticity [Kir08], both of which are more efficient. Combining those techniques while still allowing for a “streaming” protocol in which the generator and evaluator can work in parallel requires careful construction, and two novel techniques were developed for BillionYao. Chapter 3 presents a proof of security for the complete protocol that was not previously published. At the time it was published, BillionYao was the fastest 2-party malicious-model protocol, but other researchers have presented more recent work that can outperform BillionYao [WMK17, KNR+17].

Like other systems at the time, the scalability of the BillionYao system was limited by the fact that it represented functions as combinatorial circuits. The PCF 1 system presented in Chapter 4, based on the work of Kreuter, shelat, Mood, and Butler [KSMB13], solves this problem by treating the entire MPC protocol as a virtual machine runtime and allowing certain control structures to be evaluated as needed. PCF used a cross-compiling approach to translate bytecode from another compiler into a more MPC-friendly format, which in principle allows PCF to support any high-level language. To handle these unstructured bytecode inputs, new techniques for dealing with improperly nested control structures were developed. The approach of not unrolling loops or subroutine calls until the protocol is evaluated was also used by Zahr and Evans’ Obliv-C system [ZE14], but that system, unlike PCF, directly parses and compiles a specific high-level language. Other researchers have pushed the paradigm even further, combining oblivious RAM with MPC to create a secure distributed MIPS virtual machine with efficient random access memory [WGMK16].

The Geppetto system presented in Chapter 5, based on research by Costello, Fournet, Howell, Kohlweiss, Kreuter, Naehrig, Parno, and Zahur [CFH+15], follows a design similar to PCF, for succinct non-interactive arguments of knowledge (SNARKs). In this special case of MPC, one party will prove to the other that a specific algorithm was executed to compute some output. The protocol must be non-interactive, meaning that the prover should send only one message to the verifier, using a common input. Secrecy is not required, but trivial solutions are ruled out by the requirement that the size of the proof should be constant with respect to the input size. Previous work in this area, such as the Pinocchio system that Geppetto is based on [PGHR13], required the prover to compute the function as a circuit, which is both computationally inefficient and results in a very large evaluation key. Geppetto is based on the observation that because the intermediate results are not secret the encoding of the function need not be oblivious (as is required in the general 2PC setting). A naive approach would result in proof sizes that scale with the running time of the function. Succinctness is maintained by bootstrapping proofs, so that each block of the control-flow graph will first verify the proof of its preceding block. To do this efficiently, a novel technique of generating sequences of nested pairing-friendly curves, such that the field operations

1An unfortunate name collision exists here. “System PCF” refers to a particular type system in programming language theory, which we were unaware of when we chose the name “PCF” for our system.
of one curve can be evaluated in the exponent of the next, was developed. That technique, which generates
the first curves in the Barreto-Naehrig family and the rest using the Cocks-Pinch technique, is more
efficient than the MNT-curve technique of Karabina and Teske [KT08] for limited bootstrapping.

In Part II two custom protocols are presented, motivated by real-world problems at a technology
company. Those protocols have been deployed in real applications that imposed particular constraints on
performance and resource utilization. These application-imposed constraints ruled out generic protocols
and demanded novel custom protocols.

The first protocol is a variant of private set intersection referred to as private intersection-sum,
presented by Ion, Kreuter, Nergiz, Patel, Saxena, Seth, Shanahan, and Yung [IKN+17]. In this case,
one party has as input a set, and the other has a map i.e. a set of key/value pairs. The goal is to
compute the size of the intersection, using the keys from the key/value pairs, and the sum of the values
concerning to the keys in the intersection. Although this would be straightforward using garbled
circuits, the application requires large numbers intersection-sum computations every day and runs in an
environment where communication is far more expensive than computation. State-of-the-art PSI protocols
require less communication, but in many cases it remains unclear how to adapt the protocols for the
intersection-sum functionality, and among those that can be adapted the communication cost remains
higher than DDH-based protocols. Two novel DDH-based protocols are presented for this functionality,
one of which deals with an additional application-specific constraint described in Chapter 6.

The other protocol involves computing weighted averages of vectors supplied by a large number of
parties to a central aggregation service, recently published by Bonawitz, Ivanov, Kreuter, Marcedone,
McMahan, Patel, Ramage, Segal, and Seth [BIK+17]. The core protocol is similar to previous work
by Ács et al. [AC11] and by Jansen and Johnson [JJ16], but the application setting imposes unique
requirements and constraints. In this case the parties submitting vectors are unreliable and will abort with
non-negligible probability. Novel techniques are presented in Chapter 7 to handle these failures while
satisfying strict limits on communication overhead and maintaining constant round complexity.

1.2 Related Work

1.2.1 MPC Protocols

Garbled Circuits

Fairplay was an early 2PC compiler and protocol implementation [MNPS04], which served as the basis for
several subsequent research prototypes [PSSW09, MLB12a, Ker11, IS10, HKoS+10, PSS09, SVP+12,
SMBW12]. The original Fairplay protocol was not secure in the malicious model due to both the specific
details of the cut-and-choose process, which allowed cheating with non-negligible probability, as well as a
flaw observed by Mohassel and Franklin [MF06]. In addition, the security proof for the Fairplay protocol
relied on the random oracle model, which has theoretical limitations [CGH04].

Mohassel and Franklin proposed an efficient solution to the flaw in the Fairplay protocol [MF06].
Unfortunately, as Lindell and Pinkas observed, that protocol had a subtle flaw as well [LP07]. In the same
work, Lindell and Pinkas proposed a solution that was later implemented and Pinkas et al. [PSSW09].
Subsequent work on the cut-and-choose technique forms the basis for the BillionYao protocol [Kir08,
LP11b, LP11a, SS11].
The cut-and-choose technique used in BillionYao sends many copies of the entire garbled circuit to the evaluator, with some fraction being checked and the remainder being evaluated. An alternative approach known as LEGO involves sending many individual gates, some fraction of which are checked and the remainder connected by the evaluator [NO09, NST17]. Recently Kolesnikov et al. showed that a middle ground exists, in which sub-circuits are either opened or evaluated and which can be more efficient [KNR+17].

The most significant bottleneck for garbled circuits protocols is the need to send garbled gates from the generator to the evaluator. The BillionYao system uses several techniques from previous work to reduce the communication overhead. One important technique is the “free XOR” technique of Kolesnikov and Schneider [KS08], which avoids the need to send garbled XOR and XNOR gates. Another important technique is the garbled-row-reduction technique of Pinkas et al. [PSSW09], which allows the non-XOR gates to be represented by 25% fewer bytes. Subsequent to BillionYao, Zahur et al. presented a better approach, dubbed “half gates”, which reduces non-XOR gate sizes by 50% while remaining compatible with the free-XOR technique [ZRE15]. BillionYao uses the cut-and-choose technique for malicious model security, but the overhead there is reduced by 60% using Goyal et al.’s idea of sending only hashes (as commitments) of the generated circuits, and only seeds for the check circuits [GMS08].

Other Approaches and Models

The groundbreaking GMW protocol was based on using only oblivious transfer [GMW87]. Although the GMW protocol is not practical in the malicious model, recent work on OT-based MPC has shown great promise. The IPS compiler of Ishai et al. [IPS08] represented a major step forward. The IPS compiler avoids the expensive zero-knowledge proofs required by the GMW protocol, and requires constant rounds. In the two-party setting, Nielsen et al. developed another OT-based technique, which follows the approach of the GMW protocol but avoids ZKPs by using novel techniques for authenticating the output of one OT used as the input for another [NNOB11]. A critical performance technique for these protocols, which is also important for garbled circuits, is OT extension [IKNP03, ALSZ13], which allows faster symmetric operations (e.g. block ciphers) to be used for a large number of OTs, using a small number of “seed” OTs that require expensive public key operations.

OT extension has also been used in recent work on private set intersection [DCW13a, PSZ14a, RR17a]. Although some work on using garbled circuits for PSI has shown reasonable performance results [HEK12a, PSSZ15], custom protocols remain more efficient [DCT12, DCW13a, KMRS14, PSZ14a, RR17b]. Recent work on OT-based PSI has focused on using hash functions to encode one set, with OT being used to select parts of the encoding based on the other set. The approach of Dong et al., later improved by Rindal and Rosulek, uses a modified Bloom filter [DCW13a, RR17a]. Pinkas et al. presented techniques based on hash tables and Cuckoo hashing, which Rindal and Rosulek also improved [PSZ14a, RR17b]. Although these results are impressive, it is unclear how to adapt the OT-based protocols to the particular variant of PSI required in the application described in Chapter 6, where a protocol based on the DDH problem is used instead.

Some MPC protocols can be divided into an “offline” phase, which is expensive but does not depend on inputs, and a lightweight “online” phase. In the case of garbled circuits, adaptive security of a garbling scheme allows the evaluator to receive the circuit in the offline phase [BHR12a, JSW17]. Similarly, for protocols based on Beaver triples such as the SPDZ protocol [DPSZ12, KOS16], the computation of the
Beaver triples is quite expensive but can be performed independently of the inputs during the offline phase.

The online performance of protocols in the offline/online model is impressive. Unfortunately, the need to maintain state between the two phases increases the deployment complexity and imposes greater infrastructure requirements. In many cases the offline cost cannot be separated from the online cost, particularly when the MPC protocol must be run frequently, as is the case for the applications described in Part II. In the case of garbled circuits, the offline/online model forces the evaluator to store the entire garbled circuit, limiting scalability [WMK17]. The scalability of the BillionYao and PCF systems presented in Part I would not be possible in the offline/online model.

**SNARKs**

The theoretical possibility of SNARKs follows from the PCP theorem and the Fiat-Shamir heuristic [Mic00]. Bitanski et al. first suggested a construction in the standard model based on certain hash functions, and presented candidate constructions [BCCT12]. More recently, Gennaro et al. presented a more efficient construction using pairing-friendly elliptic curves [GGPR12]. Parno et al. presented a more efficient pairing-based approach with the Pinocchio system [PGHR13]. The Geppetto system presented in Chapter 5 is based on ideas from Pinocchio, but avoids the inefficient computation model used by Pinocchio.

A key idea in Geppetto is to create related pairing-friendly curves, such that the arithmetic in one curve can be performed in the exponent field of the other. A method of generating curves from the MNT family with this property was presented by Karabina and Teske [KT08], which Ben-Sasson et al. used for the SNARKs used in Zerocash [BSCTV17]. MNT curves, however, are less efficient than the BN curves used in Geppetto, and Ben-Sasson et al.'s curves achieved only an 80-bit security level. The technique used in Geppetto was suggested as a possibility by Ben-Sasson et al. but they did not present the details.

Prior to Geppetto, proof bootstrapping was described by Valient [Val08], and bootstrapping in the context of SNARKs was explored by Bitansky et al. [BCCT13]. Efficiently bootstrapping proofs motivated Ben-Sasson et al.'s use of MNT curves as well as Geppetto’s BN curves.

**1.2.2 MPC Frameworks**

As mentioned above, the Fairplay system was an early implementation of not only a 2PC protocol, but also of a supporting framework and compiler [MNPS04]. Unfortunately, while Fairplay’s circuit format and compiler were used in subsequent projects, there were limits to Fairplay’s scalability. Malka observed that the Fairplay compiler could not handle a lookup table function for only 55 elements [Mal11]. Mood et al. found similar difficulties with the Fairplay compiler and analyzed the particular bottlenecks in its engineering [MLB12b]. The compilers described in Part I are more scalable and handle much larger circuits.

Like Fairplay, the BillionYao compiler emits a boolean circuit representation of functions, which limits scalability due to the larger storage sizes needed for complex functions. One approach to solving this problem, which was used by Huang et al. and Malka [HEKM11, Mal11], is the “library” approach. In those systems, circuits are constructed on-the-fly by composing simpler circuit components. In Malka’s VMCrypt system, for example, a programmer would write Java code that instantiates a class representing a particular sub-circuit; more complex components can be created in this manner. To simplify their
library-based SMCR system, Nielsen and Schwartzbach presented SMCL, a domain-specific language that is cross-compiled to Java code that uses the SMCR library [NS07].

While library-based systems scale very well, they sacrifice automatic optimizations that compilers can perform. The reason those systems scale well is primarily due to the retention of higher-level control structures like loops, which do not exist in boolean circuits. The PCF system presented in Chapter 4 achieves the same scalability by similarly retaining such control structures. Similar approaches have been used in subsequent research. Zahur and Evans presented OblivC [ZE14], which follows a similar approach to PCF. Songhori et al.’s TinyGarble system also follows this paradigm, but with a global optimization strategy that is better than PCF’s at reducing the number of expensive non-XOR gates [LWN+15]. The same approach was used by Mood et al. in the Frigate compiler, a formally verified MPC compiler [MGC+16].

In the BillionYao system, the memory devoted to wire values is deallocated once the last gate using a wire is processed. This is tracked using a reference counting technique, with each wire’s initial count statically determined by the compiler. Henecka and Schneider refined this technique by observing that the wires can be identified by an index in the wire value array [HS13]. Instead of statically generating reference counts, their compiler statically schedules array indices that should be used to store the output value of each gate. The same observation is central to the design of PCF, but unlike PCF, Henecka and Schneider’s system still stores a complete circuit description on disk.

The design of PCF is based on a view of the entire MPC process as being a kind of virtual machine, where the “gates” are opcodes that operate on secret values. What PCF is missing is efficient random access based on secret values, which limits scalability for relatively simple functions. This can be solved by combining oblivious RAM techniques with MPC [GGH+13, WHC+14, ZWR+16, DS17]. By using ORAM for random access, Wang et al. were able to efficiently build a two-party MIPS virtual machine [WGMK16], which like PCF has, in principle, the ability to support any high-level language.

### 1.2.3 Real-World MPC Applications

One of the first real-world MPC applications was a large-scale auction system presented by Bogetoft et al. [BCD+09b]. In that case the computation was performed in the honest-but-curious model among three parties. The auction itself involves thousands of parties that supply input, in the form of secret shares, to the three parties running the MPC protocol.

The Sharemind MPC framework of Bogdanov et al. [BLW08, BNTW12] has been used in real-world MPC applications [BTW11, BKK+16]. At its core Sharemind is based on a linear secret-sharing protocol similar to the BGW protocol [BOGW88]. Unlike the BGW protocol, Sharemind uses the ring \( \mathbb{Z}_{2^{32}} \), and therefore requires a different multiplication protocol to deal with zero divisors. Recently an optimized version of protocol and new techniques for supporting floating-point arithmetic were presented by Kerik et al [KLR16]. The Sharemind system runs in the three-party setting and is secure against one corrupt party. In real-world applications of Sharemind the three parties are typically servers, similar to the auction application of Bogetoft et al., with the MPC protocol used to process data supplied by some number of clients.

Like the Sharemind applications, a recently deployed private data analytics MPC application was described by Lapets et al. [LVB+16]. In that work a custom protocol was deployed to analyze private
data about salaries at Boston-area university. Unlike Sharemind, and similar to the machine learning application described in Chapter 7, there is only one server in their application.

Another three-party honest-majority protocol used in a real-world application was recently presented by Araki et al. [AFL+16]. The motivating application for their protocol is a Kerberos service, with shares of the master secret key held by three parties. Their results demonstrate millions of AES circuits per second; this, however, is amortized, and the total latency for a single AES block remains higher than the most recent garbled circuits results. An important caveat is that in this application the parties have 10Gpbs point-to-point links, a setup which is critical in achieving such high performance.

Recently, MPC was used to generate a CRS for the Zcash cryptocurrency system [Wil]. In that case, a custom protocol was developed to meet the particular security needs of the Zcash ceremony [BSCG+15, BGG17]. Though the protocol was run over the Internet, an extra security precaution was taken by the participants: the computers used in the protocol were not directly connected to the Internet, and messages were exchanged using DVDs that were physically carried between an Internet-connected computer and the computer participating in the protocol. Such cumbersome communication was acceptable because the protocol itself had few rounds, and because the process is run infrequently.

Unlike the auction application [BCD+09b] or the Kerberos application [AFL+16], the real-world protocols described in this thesis run over the public Internet. The real-world applications of Sharemind [BTW11, BKK+16], the Zcash parameter generation system [BGG17], and the Boston salary survey [LVB+16] run over the public Internet, but only infrequently. The real-world protocols described in Part II of this thesis run many times per day every day, resulting in very strict limits on the amount of communication that can be tolerated.

The PrivCount system for conducting Tor measurements, presented by Jansen and Johnson [JJ16], operates under similarly tight constraints. Similar to the protocol from Chapter 7, PrivCount at its core computes sums using a secret-sharing approach and a central aggregator. In the PrivCount setting, however, there are fewer parties and the parties are less likely to fail than in the application described in this thesis, which admits a simpler strategy for handling aborts.
Chapter 2

Background

In their groundbreaking work, Goldwasser and Micali developed a theoretical model for analyzing secure protocols based on rigorous definitions of security [GM82]. A large body of academic work has established the feasibility and limitations of secure computation protocols for arbitrary functionalities under a variety of security models and scenarios [Yao82a, GMW87, Bea90, BMR90, DDN91, CFGN96, Can01, CLOS02, CDD+04, LLR06, GMS08, AL11, HKE12, CCL15]. Further work has been done to improve the performance of generic constructions [LP07, IPS08, KS08, NO09, PSSW09, LOP11, SS11, SS13, ZRE15].

2.1 Security Model

Intuitively, the security goal of MPC is to ensure that no party can learn more about another party’s input than what the result of the computation reveals. A complementary goal is to ensure that the output is correct, up to each party’s ability to choose its inputs prior to the execution of the protocol. In general, the privacy goal is difficult to separate from correctness.

To formally define these goals, consider a scenario where they are clearly achieved: when an additional perfectly trustworthy third party receives the inputs, performs the computation, and tells each party its specific result. This is the ideal world, which represents a formalization of the notion that the parties will learn the result of the computation. In the ideal world, each party will be allowed to do some computation to pick the inputs it sends to the trustworthy party, and then perform some computation on the response it receives to decide what to write to its output tape. The output distributions of the parties formalizes the “privacy” and “correctness” notions.

Consider a party $A_R$ attacking an actual protocol in the real world. Similar to the ideal world parties, $A_R$ will perform some computation, possibly after receiving a message from another party, to decide what to send as its first message. This will then be repeated as messages are exchanged between the parties, until the end of the protocol when parties compute their outputs.

A protocol is secure if there is a party $A_I$ in the ideal world such that the output distributions in the ideal world are indistinguishable from the distributions in the real world. Though the definition does not require it, a common proof strategy is to construct $A_I$ as a “simulator” that internally interacts with

1 A common assumption is that the parties know how large each party’s input is, though in real applications there may also be a need to hide the input size. It is possible, using FHE, to hide one party’s input size in a two-party setting [LNO12].
\( \mathcal{A}_R \). In most cases, the internal interaction is “black box” in the sense of not being dependent on \( \mathcal{A}_R \)’s “code,” though non-black-box simulators have also been studied as a technique for overcoming certain impossibility results \cite{Bar01}.

Formally there are two cases that will be considered. The first is the Honest-but-Curious case, where the parties are assumed to follow the protocol as specified. In this case the security goal is to ensure that no party will learn “more” than it can compute from the function itself. The second case is security against Malicious parties that may deviate arbitrarily from the protocol. In addition to privacy, security against malicious adversaries implies that if the protocol terminates the output is correct; in other words, the attacker cannot influence the output of honest parties beyond supply inputs to the function.

### 2.1.1 Formal Security Model

This section is based on Lindell’s tutorial on proofs in the simulation paradigm \cite{Lin17}; details on the subtleties of these definitions can be found in that tutorial.

**Definition 1** (Secure (Two-Party) Computation of \( f \) for Honest-but-Curious Parties). Let \( \pi \) be a protocol, and denote by View\( _i^\pi \)(\( x, y, k \)) the view of Party \( i \) and by Output\( _i^\pi \)(\( x, y, k \)) the output of Party \( i \), where \( k \) is the security parameter. \( \pi \) securely computes \( f \) in the honest-but-curious model if for every Party \( i \):

\[
\{(\text{Sim}_i(1^k, x_i, f(x_1, x_2)), f(x_1, x_2))\}_{x_1, x_2, k} \approx \{(\text{View}_i^\pi(x_1, x_2, k), \text{Output}_i^\pi(x_1, x_2, k))\}
\]

Definition 1 suffices when parties can be trusted to follow the protocol as it is specified. In particular, the simulator is simply given the input for Party \( i \) and the value of \( f \). A simulator that relies on this information would fail if, for example, a party deviated from the protocol by ignoring its inputs. The definition of security against malicious parties invokes the Real/Ideal paradigm, in which the simulator is treated as a party that interacts with a trusted evaluator for the functionality.

**Definition 2** (Secure (Two Party) Computation of \( f \) for Static Malicious Adversaries). Let \( \pi \) be a protocol that computes a function \( f \). We say \( \pi \) securely computes \( f \) with aborts in the presence of static malicious adversaries if for every adversary \( A \) in the real world, there exists an adversary \( S \) in the ideal world such that for every \( i \in \{1, 2\} \):

\[
\{\text{Ideal}_{f, S(z), i}(x_1, x_2, k)\}_{x, y, z, k} \approx \{\text{REAL}_{\pi, A(z), i}(x_1, x_2, k)\}_{x, y, z, k}
\]

where \( z \in \{0, 1\}^* \) is an auxiliary input and \( |x| = |y| \).

### 2.1.2 The Random Oracle Model

For the protocols described in Part II the security proofs require an idealized hash function, represented by oracle access to a random function \( \mathcal{R} : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell(k)} \). This is the Random Oracle Model, introduced by Bellare and Rogaway \cite{BR93}, which is frequently used to argue the security of practical protocols. Intuitively, this captures the assumption that the hash function has no “structural properties” that a party might take advantage of, including and beyond the collision resistance properties.

In a security proof the simulator will also reply to the oracle queries; this gives the simulator the ability to pick replies that are “convenient” or for which it has some trapdoor knowledge. For example, if the output is in some group \( G \), the simulator might record \((x, y \leftarrow g^{re}, r_y)\) i.e. the simulator will know a discrete logarithm for \( y \). This extra simulator power often allows protocols in the random oracle model to be more efficient than those in the plain model.
Unfortunately this extra simulator power and protocol efficiency carries a cost: protocols secure in the random oracle model may not be secure when instantiated with any real hash function [CGH04]. Despite this, the random oracle model is widely used in practice as a heuristic argument for security.

**Algorithm 1 Random Oracle Functionality \( \mathcal{O} \)**

**Input:** \( x \in \{0, 1\}^* \)

**Output:** \( r \in \{0, 1\}^\ell(k) \)

1: if \( \text{Prev}(x) \neq \varepsilon \) then
2: \( \text{return} \ \text{Prev}(x) \)
3: else
4: \( z \leftarrow \{0, 1\}^\ell(k) \) (uniformly sampled)
5: \( \text{Prev}(x) \leftarrow z \)
6: \text{end if}
7: \text{return} \ z

### 2.2 Computation with Circuits

To compute a function securely the computation must be oblivious to the input values; otherwise the control flow of the computation would leak some information about the input. It was shown by Pippenger and Fischer that all decision problems can be computed by an oblivious Turing Machine [PF79]. In the same work it was also shown that, for each input size, there is a Boolean circuit that computes the same function as the oblivious Turing Machine. This can be generalized to polynomials over some field, with Boolean circuits being polynomials over \( GF(2) \).

#### 2.2.1 Garbled Circuits

For protocols with only two parties, Yao’s *Garbled Circuits* construction [Yao82a] is currently the highest-performance generic protocol in terms of end-to-end running time (recent work has achieved higher performance for three parties [AFL+16]). The garbled circuits construction can be viewed as a two-step process. First, the generator party will encode the circuit according to a garbling scheme; these were described in detail by Bellare et al. [BHR12b]. In a garbling scheme, each wire in the circuit is assigned two wire keys that represent the two possible logic values of that wire; each gate will encode the wire keys corresponding to its truth table. The following is an example of a simple garbling scheme:

1. For each wire \( w_i \) in the circuit, pick two random keys \( (W_{i,0}, W_{i,1}) \).

2. For each gate, let \( (g_{0,0}, g_{0,1}, g_{1,0}, g_{1,1}) \) be the truth table, let the output wire keys be \( (W_0, W_1) \) and let the input wire keys be \( (L_0, L_1) \) and \( (R_0, R_1) \). The garbled gate is a permutation of

   \[
   \begin{array}{c|c|c}
   0 & \text{Enc}_{L_0}(\text{Enc}_{R_0}(W_{g_{0,0}})) & \text{Enc}_{L_0}(\text{Enc}_{R_1}(W_{g_{1,0}})) \\
   1 & \text{Enc}_{L_1}(\text{Enc}_{R_0}(W_{g_{1,0}})) & \text{Enc}_{L_1}(\text{Enc}_{R_1}(W_{g_{1,1}})) \\
   \end{array}
   \]

Following Yao’s early work, more efficient garbling schemes have been presented, such as Zahur et al.’s “half gates” technique [ZRE15].
The circuit can be evaluated gate-by-gate using the keys corresponding to the input wires, decrypting each gate’s output wire to get the input wire keys for subsequent gates. The evaluator party needs to receive the keys corresponding to its input wire values; for this, an oblivious transfer protocol is used. The generator can simply send the keys corresponding to its input values along with the garbled circuit.

Garbled circuit protocols perform relatively well because of their reliance on symmetric primitives e.g. block ciphers, which are typically faster than group operations required by other approaches. The only non-symmetric primitive needed for garbled circuits protocols is the OT protocol, but this can be improved using OT extension techniques [IKNP03]. Various improvements to the garbling scheme presented above are possible: XOR gates do not need to be garbled [KS08]; garbling all four truth table values is unnecessary [PSSW09, ZRE15]; the block cipher keys can be constant, improving performance on modern CPUs [BHR12b].

2.2.2 Quadratic Arithmetic Programs

**Definition 3 (QAP [PGHR13]).** A Quadratic Arithmetic Program (QAP) $Q$ over field $\mathbb{F}$ contains three sets of $\rho + 1$ polynomials $V = \{v_k(x)\}$, $W = \{w_k(x)\}$, $Y = \{y_k(x)\}$, for $k \in \{0 \ldots \rho\}$, and a target polynomial $d(x)$. Suppose $F$ is a function that takes as input $n$ elements of $\mathbb{F}$ and outputs $n'$ elements, for a total of $N = n + n'$ I/O elements. Then we say that $Q$ computes $F$ if:

$$p(x) = \left( v_0(x) + \sum_{k=1}^{\rho} c_k \cdot v_k(x) \right) \cdot \left( w_0(x) + \sum_{k=1}^{\rho} c_k \cdot w_k(x) \right) - \left( y_0(x) + \sum_{k=1}^{\rho} c_k \cdot y_k(x) \right).$$

In other words, there must exist some polynomial $h(x)$ such that $h(x) \cdot d(x) = p(x)$. The size of $Q$ is $\rho$, and the degree is the degree of $d(x)$.

2.3 Cryptographic Primitives

2.3.1 Diffie-Hellman Protocols

**Assumption 1 (Decisional Diffie-Hellman assumption (DDH)).** [DH76] Let $G(\lambda)$ be a family of groups parameterized by security parameter $\lambda$. For every probabilistic adversary $M$ that runs in time polynomial in $\lambda$, we define the advantage of $M$ to be:

$$\left| Pr[M(\lambda, g, g^a, g^b, g^{ab}) = 1] - Pr[M(\lambda, g, g^a, g^b, c) = 1] \right| - \frac{1}{2}$$

Where the probability is over a random choice $G$ from $G(\lambda)$, ran generator $g$ of $G$, random $a, b, c \in [1, |G|]$ and the randomness of $M$. The Decisional Diffie Hellman assumption holds for $G$ if for every such $M$, there exists a negligible function $\epsilon$ such that the advantage of $M$ is bounded by $\epsilon(\lambda)$.

The security of the famous Diffie-Hellman Key Exchange protocol can be reduced to the DDH assumption. Many other protocols can have their security proved by reduction to the DDH problem. One important example is the 1-of-2 OT protocol by Naor and Pinkas [NP01], presented in Figure 2.1.
Chooser’s Message  Sample $a, b, c_{1-x} \leftarrow \mathbb{Z}_q$ and set $c_x = ab$. Send $(g^a, g^b, g^{c_0}, g^{c_1})$ to the Sender.

Sender’s Message  Check $g^{c_0} \neq g^{c_1}$. Sample $r_0, r_1, s_0, s_1 \leftarrow \mathbb{Z}_q$ and send $(g^{a r_0}, g^{c_0 r_0}, g^{s_0}, g^{a s_1}, g^{r_1}, m_1 g^{c_{1s_1}} g^{br_1})$ to the chooser.

Chooser Decoding  Compute $(m_x g^{cx} g^{br_x})/(g^{as_x} g^{br_x})^b$ to receive $m_x$.

Figure 2.1: The 1-of-2 OT protocol of Naor and Pinkas [NP01].

Also important is the Schnorr proof-of-knowledge protocol [Sch90], which forms the basis for a useful family of zero-knowledge proofs [CS97] and for the Schnorr Signature System [Seu12], shown in Figure 2.2.

Prover Commitment  The prover samples $r \leftarrow \mathbb{Z}_q$ and sends $g^r$ to the verifier.

Verifier Challenge  The verifier samples $c \leftarrow \mathbb{Z}_q$ and sends $c$ to the prover.

Prover Response  The prover sends $z = cx + r$ to the verifier.

Verifier Check  The verifier checks $g^z = (g^x)^c g^r$.

Figure 2.2: The Schnorr protocol, a proof-of-knowledge of the discrete logarithm of $g^x$ [Sch90].

Another useful protocol based on the DDH assumption is the Private Set Intersection protocol by Huberman et al. [HFH99a], presented in Figure 2.3.

Figure 2.3: The simple DDH-PSI protocol.
2.3.2 Pairing-Friendly Elliptic Curves

While the constructions based on the DDH assumption described above have many useful extensions, they are limited by the inability to multiply unknown discrete logarithms. In other words, DDH protocols are limited to linear or affine functions “in the exponent.” Although higher-degree functionalities in general remain an active research topic [GGH15, CLLT16], quadratic functions can be supported using bilinear pairings. Such operations are possible in certain elliptic and hyperelliptic curve groups.

For an elliptic curve $E$ defined over a finite field $\mathbb{F}_q$, the points on $E[\mathbb{F}_q]$ form a group. More generally, for every extension field $\mathbb{F}_{q^k}$, points on $E[\mathbb{F}_{q^k}]$ form a group, with $E[\mathbb{F}_q]$ being one particular subgroup.

For sufficiently high-degree extensions, there is a function $e : G_1 \times G_2 \rightarrow G_T$ where $G_1, G_2 \subseteq E[\mathbb{F}_{q^k}]$, $G_T \subseteq \mathbb{F}_{q^k}^*$ such that $\forall x \in G_1, y \in G_2, e(x^a, y^b) = e(x, y)^{ab}$. This is known as a bilinear pairing.

The minimum $k$ for which bilinear pairings exist is known as the embedding degree of the curve. Miller’s algorithm can be used to compute bilinear pairings efficiently for small enough embedding degrees [Mil86].

Typically the embedding degree of a curve will be so large that pairings are infeasible to compute. Many techniques for finding curves with low embedding degree have been published [CFA+12].

The discrete logarithm problem on pairing-friendly curves is believed to be computationally difficult, and it is believed that the CDH assumption holds in $G_1, G_2$ and $G_T$. Note that this implies that “inverting” a pairing is hard; in other words, given $P, e(P, Q)$ it is hard to find $Q$, and given $Q, e(P, Q)$ it is hard to find $P$.

It is often desirable to state hardness assumptions as decision problems. One such assumption related to pairings is the External Diffie-Hellman Assumption (XDH) [KU16]:

**Assumption 2** (External Diffie Hellman Assumption). *Let the CDH problem be hard in $G_1$ and $G_2$. The XDH assumption is that DDH is hard in $G_1$.***

The XDH assumption does not hold for pairings in which an efficiently computable isomorphism $\phi : G_1 \rightarrow G_2$ exists; the DDH problem on $G_1$ could be solved in the $G_T$ group using $\tilde{e}(P, Q) = e(P, \phi(Q))$ in that case. Fortunately, more efficient state-of-the-art pairing-friendly curves do not appear to have such maps [KU16].

2.3.3 Homomorphic Encryption

Homomorphic Encryption schemes are encryption schemes with the property that for some class of functions $\mathcal{F}$, there exists an operation $\text{Eval}$ such that $\forall f \in \mathcal{F} \ Dec(\text{Eval}(f, \text{Enc}(m_1), \text{Enc}(m_2), \ldots, \text{Enc}(m_n))) = f(m_1, m_2, \ldots, m_n)$. In other words, a homomorphic encryption scheme allows for computation over encrypted data. Fully Homomorphic Encryption (FHE) schemes are those where $\mathcal{F}$ contains all efficiently computable functions.

**ElGamal Encryption**

The ElGamal encryption scheme is CPA-secure under the DDH assumption, and supports homomorphic group operations (denoted $\text{ElGl.Mul}$ below). If the plaintext space is small, addition in the exponent can also be supported, but decryption in this case requires inverting a discrete logarithm. Using the identity element of the group, $\text{ElGl.Mul}$ can be used to re-randomize a ciphertext.
Definition 4 (ElGamal Encryption). The ElGamal encryption scheme [ElG85] is an additively homomorphic encryption scheme, consisting of the following probabilistic polynomial-time algorithms:

ElGl.Gen  Given a security parameter \( \lambda \), ElGl.Gen(\( \lambda \)) returns outputs a public-private key pair \((pk, sk)\), and specifies a message space \( \mathcal{M} \).

ElGl.Enc  Given the public key \( pk \) and a plaintext message \( m \in \mathcal{M} \), one can compute a ciphertext \( \text{ElGl.Enc}(pk, m) \), a Paillier encryption of \( m \) under \( pk \). (In the following chapters this is shortened to just \( \text{ElGl}(m) \) when \( pk \) is clear from the context).

ElGl.Dec  Given the secret key \( sk \) and a ciphertext \( \text{ElGl.Enc}(pk, m) \), one can run \( \text{ElGl.Dec} \) to recover the plaintext \( m \).

ElGl.Mul  Given the public key \( pk \) and a set of ciphertexts \( \{\text{ElGl.Enc}(pk, m_i)\} \) encrypting messages \( \{m_i\} \), one can homomorphically compute a ciphertext encrypting the sum of the underlying messages:

\[
\text{ElGl.Enc}(pk, \sum_i m_i) = \text{ElGl.Mul}(\{\text{ElGl.Enc}(pk, m_i)\})
\]

Paillier Encryption

The Paillier encryption system supports homomorphic additions, or more generally, affine functions.

Definition 5 (Paillier Homomorphic Encryption). The Paillier encryption scheme [Pai99] is an additively homomorphic encryption scheme, consisting of the following probabilistic polynomial-time algorithms:

Pai.Gen  Given a security parameter \( \lambda \), Pai.Gen(\( \lambda \)) returns outputs a public-private key pair \((pk, sk)\), and specifies a message space \( \mathcal{M} \).

Pai.Enc  Given the public key \( pk \) and a plaintext message \( m \in \mathcal{M} \), one can compute a ciphertext \( \text{Pai.Enc}(pk, m) \), a Paillier encryption of \( m \) under \( pk \). (In the following chapters this is shortened to just \( \text{Pai}(m) \) when \( pk \) is clear from the context).

Pai.Dec  Given the secret key \( sk \) and a ciphertext \( \text{Pai.Enc}(pk, m) \), one can run \( \text{Pai.Dec} \) to recover the plaintext \( m \).

Pai.Sum  Given the public key \( pk \) and a set of ciphertexts \( \{\text{Pai.Enc}(pk, m_i)\} \) encrypting messages \( \{m_i\} \), one can homomorphically compute a ciphertext encrypting the sum of the underlying messages:

\[
\text{Pai.Enc}(pk, \sum_i m_i) = \text{Pai.Sum}(\{\text{Pai.Enc}(pk, m_i)\})
\]

A useful extension of Paillier encryption was presented by Damgård and Jurik [DJ01] that allows encryption of arbitrarily large integers.
Part I

Generic Protocols
Chapter 3

Billion-Gate Secure Computation with Malicious Adversaries

This chapter describes the BillionYao system, based on the work of Kreuter et al. [KSS12]. BillionYao is a generic protocol for two-party computation in the malicious model, and a larger system that includes a circuit compiler and runtime library. The goal of BillionYao is to support large circuits, with billions of gates, in the malicious model. To support such large circuits, the BillionYao system carefully manages computational resources in both the core protocol and the larger framework.

3.1 Introduction

Andrew Yao’s groundbreaking Garbled Circuits construction [Yao82a] is an important early result on secure two-party computation. Yao’s protocol involves representing the function as a boolean circuit and having one party (called the generator) encrypt the circuit in such a way that it can be selectively decrypted by the other party (called the evaluator) to compute the output, a process called garbling. In particular, oblivious transfers are used for the evaluator to obtain a subset of the decryption keys that are needed to compute the output of the function.

Yao’s original protocol ensures the privacy of each party’s input and the correctness of the output under the semi-honest model, in which both parties follow the protocol honestly. This model has been the basis for several scalable secure computation systems [BS05c, LP00, JKS08, HKoS+10, GHS12, Mal11, HEKM11]. It is conceivable, however, that one of the parties may deviate from the protocol in an attempt to violate privacy or correctness. Bidders may attempt to manipulate the auction output in their favor; spies may attempt to obtain sensitive information; and a computer being used for secure computation may be infected with malware. Securing against malicious participants, who may deviate arbitrarily from pre-agreed instructions, in an efficient manner is of more practical importance.

Prior to the BillionYao system there had been several attempts on practical systems with security against active, malicious adversaries. Lindell and Pinkas presented an approach based on garbled circuits that uses the cut-and-choose technique [LP07], with an implementation of this system having been given by Pinkas et al. [PSSW09]. Nielsen et al. presented the LEGO+ system [NNOB11], which uses efficient oblivious transfers and authenticated bits to enforce honest behaviors from participants. shelat and Shen proposed a hybrid approach that integrates sigma protocols into the cut-and-choose technique [SS11].
The protocol compiler presented by Ishai, Prabhakaran, and Sahai [IPS08] also uses an approach based on oblivious transfer, and was implemented by Lindell, Oxman, and Pinkas [LOP11]. In all these cases, AES was used as a benchmark for performance tests.

Protocols for general multiparty computation with security against a malicious majority have also been presented. Canetti et al. gave a construction of a universally composable protocol in the common reference string model [CLOS02]. The protocol compiler of Ishai et al., mentioned above, can be used to construct a multiparty protocol with security against a dishonest majority in the UC model [IPS08]. Bendlin et al. showed a construction based on homomorphic encryption [BDOZ11], which was improved upon by Damgård et al. [DPSZ12]; these protocols were also proved secure in the UC model, and thus require additional setup assumptions. The protocol of Damgård et al. (dubbed “SPDZ” and pronounced “speedz”) is based on a preprocessing model, which improves the amortized performance. Damgård et al. presented an implementation of their protocol, which could evaluate the function \((x \times y) + z\) in about 3 seconds with a 128 bit security level, but with an amortized time of a few milliseconds.

This chapter presents a scalable two-party secure computation system which guarantees privacy and correctness in the presence of a malicious party. The BillionYao system can handle circuits with hundreds of millions or even billions of gates, while requiring relatively modest computing resources. BillionYao follows the Fairplay framework, allowing general purpose secure computation starting from a high level description of a function. BillionYao, however, has numerous technical advantages over the Fairplay system, both in our compiler and in the secure computation protocol. Unlike previous work, the experimental results presented below do not rely solely on AES circuits as our benchmark; the goal of these experiments was to evaluate circuits that are orders of magnitude larger than AES in the malicious model, and AES was used only as a comparison with previous work. The protocol is proved secure only assuming circular 2-correlation robust hash functions and the hardness of the elliptic curve discrete logarithm problem, and requires neither additional setup assumptions nor the random oracle model.

**System Framework**  The system is based on Yao’s garbled circuit [Yao82a] protocol for securely computing functions in the presence of semi-honest adversaries, and shelat and Shen’s cut-and-choose-based transformation [SS11] that converts Yao’s garbled circuit protocol into one that is secure against malicious adversaries. From there, the protocol is modified to use Ishai et al.’s oblivious transfer extension [IKNP03] that has efficient amortized computation time for oblivious transfers secure against malicious adversaries, and Lindell and Pinkas’ random combination technique [LP07] that defends against selective failure attacks. Kiraz’s randomized circuit technique [Kir08] to guarantee that the generator gets either no output or an authentic output is also used as a performance improvement.

**Optimization Techniques**  For garbled circuit generation and evaluation, Kolesnikov and Schneider’s free-XOR technique is used to minimize the computation and communication cost for XOR gates in the circuit [KS08]. Also used is Pinkas et al.’s garbled-row-reduction technique that reduces the communication cost for \(k\)-fan-in non-XOR gates by \(1/2^k\) [PSSW09], which means at least a 25% communication saving in our system since we only have gates of 1-fan-in or 2-fan-in. A further optimization implemented in BillionYao is Goyal et al.’s technique for reducing communication, which proceeds as follows: during the cut-and-choose step, the check circuits are given to the evaluator by revealing the random seeds used to produce them rather than the check circuits themselves [GMS08]. Combined with the 60%–40%
check-evaluation ratio proposed by Shelat and Shen [SS11], Goyal et al.’s technique provides a near 60% saving in communication.

Circuit-Level Parallelism An important technique used to improve the running time in BillionYao follows from the observation that Shelat and Shen’s cut-and-choose protocol is embarrassingly parallel. Taking advantage of this, however, requires careful engineering in order to achieve better performance while maintaining security. All computation-intensive operations such as oblivious transfers or circuit construction are parallelized by splitting the generator-evaluator pair into many “slave” pairs. Each of the slave pairs works on an independently generated copy of the circuit in a parallel but synchronized manner, as some synchronization is required for Shelat and Shen’s protocol [SS11] to be secure.

Computation Complexity For the computation time of a secure computation, there are two main bottlenecks: the input processing time $I$ (due to oblivious transfers) and the circuit processing time $C$ (due to garbled circuit construction and evaluation). In the semi-honest model, the system’s computation time is simply $I + C$. Security in the malicious model, however, requires several extra checks. In the first instantiation of our system, through heavy use of circuit-level parallelism, our system needs roughly $I + 2C$ to compute hundreds of copies of the circuit. Thus when the circuit size is sufficiently larger than the input size, the BillionYao system (secure in the malicious model) needs roughly twice as much computation time as that needed by the original Yao protocol (secure in the semi-honest model). This represents a large improvement over prior work [SS11, PSSW09] which needed 100x more time than the semi-honest Yao. In the second instantiation of this scheme, the prototype was able to achieve $I + C$ computation time, at the cost of moderately more communication overhead.

Large Circuits In the Fairplay system, a garbled circuit is fully constructed before being sent over a network for the other party to evaluate. This approach is particularly problematic when hundreds of copies of a garbled circuit are needed against malicious adversaries. Huang et al. [HEKM11] pointed out that keeping the whole garbled circuit in memory is unnecessary, and that instead, the generation and evaluation of garbled gates could be conducted in a “pipelined” manner. Consequently, not only do both parties spend less time idling, only a small number of garbled gates need to reside in memory at one time, even when dealing with large circuits. However, this pipelining idea does not work trivially with other optimization techniques for the following two reasons:

- The cut-and-choose technique requires the generator to finish constructing circuits before the coin flipping (which is used to determine check circuits and evaluation circuits), but the evaluator cannot start checking or evaluating before the coin flipping. A naive approach would ask the evaluator to hold the circuits and wait for the results of the coin flipping before she proceeds to do her jobs. When the circuit is of large size, keeping hundreds of copies of such a circuit in memory is undesirable.

- Similarly, the random seed checking technique [GMS08] requires the generator to send the hash for each garbled circuit, and later on send the random seeds for check circuits so that the communication for check circuits is vastly reduced. Note that the hash for an evaluation circuit is given away before the garbled circuit itself. However, a hash is calculated only after the whole circuit is generated. So the generation-evaluation pipelining cannot be applied directly.
BillionYao, however, integrates this pipelining idea with the optimization techniques mentioned above, and is capable of handling circuits of billions of gates.

**AES-NI**  Besides the improvements by the algorithmic means, Intel’s Advanced Encryption Standard Instructions (AES-NI) were used in the prototype implementation. While the encryption is previously suggested to be

$$\text{Enc}_{X,Y}(Z) = H(X||Y) \oplus Z$$

in the literature [KS08, CKKZ12], where $H$ is a 2-circular correlation robust function instantiated either with SHA-1 [HEKM11] or SHA-256 [PSSW09]. BillionYao uses an alternative $\text{Enc}_{X,Y}^{k}(Z) = \text{AES-256}_{X||Y}(k) \oplus Z$, where $k$ is the index of the garbled gate. With the help of the latest instruction set, an AES-256 operation could take as little as 30% of the time for SHA-256. Since this operation is heavily used in circuit operations, with the help of AES-NI instructions, we are able to reduce the circuit computation time $C$ by at least 20%. Bellare’s work on garbling schemes showed more formally that this is indeed secure [BHR12b].

**Performance**  Using AES as a benchmark, the KSS12 system represents a clear improvement of previous work. Results in the semi-honest model were reported by Huang et al.[HEKM11], whose system needed 1.3 seconds (where $I = 1.1$ and $C = 0.2$) to complete a block of secure AES computation. The fastest previously known system in the malicious model was proposed by Nielson et al.[NNOB11] and has an amortized performance 1.6 seconds per block (or more precisely, $I = 79$ and $C = 6$ for 54 blocks). The KSS12 system provides security in the malicious model and needs $I.A = I + 2C$, where $I = 1.0$ and $C = 0.2$) seconds per block. Note that both the prior systems require the full power of a random oracle, while KSS12 requires a weaker cryptographic primitive, 2-circular correlation robust functions, which was recently shown to be sufficient to prove the security of the free-XOR technique. It should also be noted that our system benefits greatly from parallel computation, which was not tested for LEGO+.

**Scalable Circuit Compiler**  One of the major bottlenecks that prevented large-scale secure computation using previous frameworks was the lack a scalable compiler. Prior tools could barely handle circuits with 50,000 gates, requiring significant computational resources to compile such circuits. While this is just enough for an AES circuit, it is not enough for the large circuits that we evaluate in this paper.

In the BillionYao system a scalable boolean circuit compiler was implemented that can be used to generate circuits with billions of gates, with moderate hardware requirements. This compiler performs some simple but highly effective optimizations, and tends to favor XOR gates. The toolchain is flexible, allowing for different levels of optimizations and can be parameterized to use more memory or more CPU time when building circuits.

As a first demonstration that this compiler advanced the state of the art, observe that it automatically generates a smaller boolean circuit for the AES cipher than the hand-optimized circuit reported by Pinkas et al. [PSSW09]. AES plays an important role in secure computation, and oblivious AES evaluation can be used as a building block in cryptographic protocols. Not only is it one of the most popular building blocks in cryptography and real life security, it is often used as a benchmark in secure computation. With the textbook algorithm, the well-known Fairplay compiler can generate an AES circuit that has 15,316 non-XOR gates. Pinkas et al. were able to develop an optimized AES circuit that has 11,286 non-XOR
gates. By applying an efficient S-box circuit [BP10] and using the BillionYao compiler, we were able to construct an AES circuit that has 9,100 non-XOR gates. As a result, this AES circuit only needs 59% and 81% of the communication needed by the other two, respectively.

Most importantly, using BillionYao and its scalable compiler, we are able to run experiments on circuits with sizes in the range of billions of gates. These circuits include 256-bit RSA (266,150,119 gates) and 4095x4095-bit edit distance (5,901,194,475). As the circuit size grows, resource management becomes crucial. A circuit of billions of gates can easily result in several GB of data stored in memory or sent over the network. Special care was required to handle these difficulties.

**Chapter Organization** The organization of this chapter is as follows. A variety of security decisions and optimization techniques will be covered in Section 3.2 and Section 3.3, respectively. Then, our system, including a compiler, will be introduced in Section 3.5. Finally, the experimental results are presented in Section 3.6.

### 3.2 Security Techniques

The Yao protocol, while efficient, assumes honest behavior from both parties. To achieve security in the malicious model, it is necessary to enforce honest behavior. The cut-and-choose technique is one of the most efficient methods in the literature and is used in our system. Its main idea is for the generator to prepare multiple copies of the garbled circuit with independent randomness, and the evaluator picks a random fraction of the received circuits, whose randomness is then revealed. If any of the chosen circuits (called check circuits) is not consistent with the revealed randomness, the evaluator aborts; otherwise, she evaluates the remaining circuits (called evaluation circuits) and takes the majority of the outputs, one from each evaluation circuit, as the final output.

The intuition is that to pass the check, a malicious generator can only sneak in a few faulty circuits, and the influence of these (supposedly minority) faulty circuits will be eliminated by the majority operation at the end. On the other hand, if a malicious generator wants to manipulate the final output, she needs to construct faulty majority among evaluation circuits, but then the chance that none of the faulty circuits is checked will be negligible. So with the help of the cut-and-choose method, a malicious generator either constructs many faulty circuits and gets caught with high probability, or constructs merely a few and has no influence on the final output.

However, the cut-and-choose technique is not a cure-all. Several subtle attacks have been reported and would be a problem if not properly handled. These attacks include the generator’s input inconsistency attack, the selective failure attack, and the generator’s output authenticity attack, which are discussed in the following sections. Note that in this section, $n$ denotes the input size and $s$ denotes the number of copies of the circuit.

**Generator’s Input Consistency** Recall that in the cut-and-choose step, multiple copies of a circuit are constructed and then evaluated. A malicious generator is therefore capable of providing altered inputs to different evaluation circuits. It has been shown that for some functions, there are simple ways for the generator to extract information about the evaluator’s input [LP07]. For example, suppose both parties agree to compute the inner-product of their input, that is, $f([a_2, a_1, a_0], [b_2, b_1, b_0]) \mapsto a_2b_2 + a_1b_1 + a_0b_0$ where $a_i$ and $b_i$ is the generator’s and evaluator’s $i$-th input bit, respectively. Instead of providing
[\mathbf{a}_2, \mathbf{a}_1, \mathbf{a}_0] to all evaluation circuits, the generator could send \([1, 0, 0], [0, 1, 0], \text{ and } [0, 0, 1]\) to different copies of the evaluation circuits. After the majority operation from the cut-and-choose technique, the generator learns major\((\mathbf{b}_2, \mathbf{b}_1, \mathbf{b}_0)\), the majority bit in the evaluator’s input, which is not what the evaluator agreed to reveal in the first place.

There exist several approaches to deter this attack. Mohassel and Franklin [MF06] proposed the equality-checker that needs \(O(ns^2)\) commitments to be computed and exchanged. Lindell and Pinkas [LP07] developed an approach that also requires \(O(ns^2)\) commitments. Later, Lindell and Pinkas [LP11b] proposed a pseudorandom synthesizer that relies on efficient zero-knowledge proofs for specific hardness assumptions and requires \(O(ns)\) group operations. Shelat and Shen [SS11] suggested the use of malleable claw-free collections, which also uses \(O(ns)\) group operations, but they showed that witness-indistinguishability suffices, which is more efficient than zero-knowledge proofs by a constant factor.

In our system, we incorporate the malleable claw-free collection approach because of its efficiency. Although the commitment-based approaches can be implemented using lightweight primitives such as collision-resistant hash functions, they incur high communication overhead for the extra complexity factor \(s\), that is, the number of copies of the circuit. On the other hand, the group-based approach could be more computationally intensive, but this discrepancy is compensated again due to the parameter \(s\).\(^1\) Hence, with similar computation cost, group-based approaches enjoy lower communication overhead.

Selective Failure  A more subtle attack is *selective failure* [MF06, KS06]. A malicious generator could use inconsistent keys to construct the garbled gate and OT so that the evaluator’s input can be inferred from whether or not the protocol completes. In particular, a cheating generator could assign \((K_0, K_1)\) to an input wire in the garbled circuit while using \((K_0, K_1^*)\) instead in the corresponding OT, where \(K_1 \neq K_1^*\). As a result, if the evaluator’s input is 0, she learns \(K_0\) from OT and completes the evaluation without complaints; otherwise, she learns \(K_1^*\) and gets stuck during the evaluation. If the protocol expects the evaluator to share the result with the generator at the end, the generator learns whether or not the evaluation failed, and therefore, the evaluator’s input is leaked.

Lindell and Pinkas [LP07] proposed the random input replacement approach that involves replacing each of the evaluator’s input bits with an XOR of \(s\) additional input bits, so that whether the evaluator aborts due to a selective failure attack is almost independent (up to a bias of \(2^{1-s}\)) of her actual input value. Both Kiraz [Kir08] and Shelat and Shen [SS11] suggested a solution that exploits committing OTs so that the generator commits to her input for the OT, and the correctness of the OTs can later be checked by opening the commitments during the cut-and-choose. Lindell and Pinkas [LP11b] also proposed a solution to this problem using cut-and-choose OT, which combines the OT and the cut-and-choose steps into one protocol to avoid this attack.

Our system is based on the random input replacement approach due to its scalability. It is a fact that the committing OT or the cut-and-choose OT does not alter the circuit while the random input replacement approach inflates the circuit by \(O(sn)\) additional gates. However, it has been shown that \(\max(4n, 8s)\) additional gates suffice [PSSW09]. Moreover, both the committing OT and the cut-and-choose OT require

\(^1\)To give concrete numbers, with an Intel Core i5 processor and 4GB DDR3 memory, a SHA-256 operation (from OpenSSL) requires 1,746 cycles, while a group operation (160-bit elliptic curve from the PBC library with preprocessing) needs 322,332 cycles. It is worth-mentioning that \(s\) is at least 256 in order to achieve security level \(2^{-80}\). The gap between a symmetric operation and an asymmetric one becomes even smaller when modern libraries such as RELIC are used instead of PBC.
The random input replacement approach needs only $O(s)$ group operations. Furthermore, we observe that the random input replacement approach is in fact compatible with the OT extension technique. Therefore, we were able to build our system which has the group operation complexity independent of the evaluator’s input size, and as a result, our system is particularly attractive when handling a circuit with a large evaluator input.

**Generator’s Output Authenticity** It is not uncommon that both the generator and evaluator receive outputs from a secure computation, that is, the goal function is $f(x, y) = (f_1, f_2)$, where the generator with input $x$ gets output $f_1$, and the evaluator with input $y$ gets $f_2$. In this case, the security requires that both the input and output are hidden from each other. In the semi-honest setting, the straightforward solution is to let the generator choose a random number $c$ as an extra input, convert $f(x, y) = (f_1, f_2)$ into a new function $f^*((x, c), y) = (\lambda, (f_1 \oplus c, f_2))$, run the original Yao protocol for $f^*$, and instruct the evaluator to pass the encrypted output $f_1 \oplus c$ back to the generator, who can then retrieve her real output $f_1$ with the secret input $c$ chosen in the first place. However, the situation gets complicated when either of the participants could potentially be malicious. In particular, a malicious evaluator might claim an arbitrary value to be the generator’s output coming from the circuit evaluation. Note that the two-output protocols we consider are not fair since the evaluator always learns her own output and may refuse to send the generator’s output. However, they can satisfy the notion that the evaluator cannot trick the generator into accepting arbitrary output.

Many approaches have been proposed to ensure the generator’s output authenticity. Lindell and Pinkas [LP07] proposed a solution similar to the aforementioned solution in the semi-honest setting, where the goal function is modified to compute $f_1 \oplus c$ and its MAC so that the generator can verify the authenticity of her output. This approach incurs a cost of adding $O(n^2)$ gates to the circuit. Kiraz [Kir08] presented a two-party computation protocol in which a zero knowledge proof of size $O(s)$ is conducted at the end. Sahar and Shen [SS11] suggested a signature-based solution which, similar to Kiraz’s, adds $n$ gates to the circuit, and requires a proof of size $O(s + n)$ at the end. However, they observed that witness-indistinguishable proofs are sufficient.

Lindell and Pinkas’ approach, albeit straightforward, might introduce greater communication overhead than the description function itself. We therefore employ the approach that takes the advantages of the remaining two solutions. In particular, we implement Kiraz’s approach (smaller proof size), but only a witness-indistinguishable proof is performed (weaker security property).

### 3.3 Performance Techniques

Yao’s garbled circuit technique has been studied for decades. It has drawn significant attention for its simplicity, constant round complexity, and computational efficiency (since circuit evaluation only requires fast symmetric operations). The fact that it incurs high communication overhead has provoked interest that has led to the development of fruitful results.

In this section, we will first briefly present the Yao garbled circuit, and then discuss the optimization techniques that greatly reduce the communication cost while maintaining the security. These techniques include free-XOR, garbled row reduction, random seed checking, and large circuit pre-processing. In

---

2Here $f_1$ and $f_2$ are short for $f_1(x, y)$ and $f_2(x, y)$ for simplicity.
addition to these original ideas, practical concerns involving large circuits and parallelization will be addressed.

### 3.3.1 Baseline Yao’s Garbled Circuit

Given a circuit that consists of 2-fan-in boolean gates, the generator constructs a garbled version as follows: for each wire \( w \), the generator picks a random permutation bit \( \pi_w \in \{0, 1\} \) and two random keys \( w_0, w_1 \in \{0, 1\}^{k-1} \). Let \( W_0 = w_0 \| \pi_w \) and \( W_1 = w_1 \| (\pi_w \oplus 1) \), which are associated with bit value 0 and 1 of wire \( w \), respectively. Next, for gate \( g \in \{ f \mid f : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\} \} \) that has input wire \( x \) with \( (X_0, X_1, \pi_x) \), input wire \( y \) with \( (Y_0, Y_1, \pi_y) \), and output wire \( z \) with \( (Z_0, Z_1, \pi_z) \), the garbled truth table for \( g \) has four entries:

\[
\text{GTT}_g = \begin{cases} 
\text{Enc}(X_0 \oplus \pi_x, Y_0 \oplus \pi_z, Z_g(0 \oplus \pi_x, 0 \oplus \pi_y)) \\
\text{Enc}(X_0 \oplus \pi_x, Y_1 \oplus \pi_z, Z_g(0 \oplus \pi_x, 1 \oplus \pi_y)) \\
\text{Enc}(X_1 \oplus \pi_x, Y_0 \oplus \pi_z, Z_g(1 \oplus \pi_x, 0 \oplus \pi_y)) \\
\text{Enc}(X_1 \oplus \pi_x, Y_1 \oplus \pi_z, Z_g(1 \oplus \pi_x, 1 \oplus \pi_y)). 
\end{cases}
\]

\( \text{Enc}(K, m) \) denotes the encryption of message \( m \) under key \( K \). Here the encryption key is a concatenation of two labels, and each label is a random key concatenated with its permutation bit. Intuitively, \( \pi_x \) and \( \pi_y \) permute the entries in \( \text{GTT}_g \) so that for \( i_x, i_y \in \{0, 1\} \), the \( (2i_x + i_y) \)-th entry represents the input pair \((i_x \oplus \pi_x, i_y \oplus \pi_y)\) for gate \( g \), in which case the label associated with the output value \( g(i_x \oplus \pi_x, i_y \oplus \pi_y) \) could be retrieved. More specifically, to evaluate the garbled gate \( \text{GTT}_g \), suppose \( X \| b_x \) and \( Y \| b_y \) are the retrieved labels for input wire \( x \) and wire \( y \), respectively, the evaluator will use \( X \| b_x \| Y \| b_y \) to decrypt the \( (2b_x + b_y) \)-th entry in \( \text{GTT}_g \) and retrieve label \( Z \| b_z \), which is then used to evaluate the gates at the next level. The introduction of the permutation bit helps to identify the correct entry in \( \text{GTT}_g \), and thus, only one, rather than all, of the four entries will be decrypted.

### 3.3.2 Free-XOR

Kolesnikov and Schneider [KS08] proposed the free-XOR technique that aims for removing the communication cost and decreasing the computation cost for XOR gates.

The idea is that the generator first randomly picks a global key \( R \), where \( R = r \| 1 \) and \( r \in \{0, 1\}^{k-1} \). This global key has to be hidden from the evaluator. Then for each wire \( w \), instead of picking both \( W_0 \) and \( W_1 \) at random, only one is randomly chosen from \( \{0, 1\}^k \), and the other is determined by \( W_b = W_1 \oplus b \oplus R \). Note that \( \pi_w \) remains the rightmost bit of \( W_0 \). For an XOR gate having input wire \( x \) with \( (X_0, X_0 \oplus R, \pi_x) \), input wire \( y \) with \( (Y_0, Y_0 \oplus R, \pi_y) \), and output wire \( z \), the generator lets \( Z_0 = X_0 \oplus Y_0 \) and \( Z_1 = Z_0 \oplus R \). Observe that

\[
X_0 \oplus Y_1 = X_1 \oplus Y_0 = X_0 \oplus Y_0 \oplus R = Z_0 \oplus R = Z_1
\]
\[
X_1 \oplus Y_1 = X_0 \oplus R \oplus Y_0 \oplus R = X_0 \oplus Y_0 = Z_0.
\]

This means that while evaluating an XOR gate, XORing the labels for the two input wires will directly retrieve the label for the output wire. So no garbled truth table is needed, and the cost of evaluating an XOR gate is reduced from a decryption operation to a bitwise XOR.
This technique is only secure when the encryption scheme satisfies certain security properties. The solution provided by the authors is

\[ \text{Enc}(X || Y, K) = H(X || Y) \oplus Z, \]

where \( H : \{0, 1\}^{2k} \mapsto \{0, 1\}^k \) is a random oracle. Recently, Choi et al. [CKKZ12] have further shown that it is sufficient to instantiate \( H(\cdot) \) with a weaker cryptographic primitive, 2-circular correlation robust functions. Our system instantiates this primitive with \( H(\cdot) = \text{SHA-256}(X || Y) \). However, when AES-NI instructions are available, our system instantiates it with \( H_k(X || Y) = \text{AES-256}(X || Y, k) \), where \( k \) is the gate index.

### 3.3.3 Garbled Row Reduction

The GRR (Garbled Row Reduction) technique suggested by Pinkas et al. [PSSW09] is used to reduce the communication overhead for non-XOR gates. In particular, it reduces the size of the garbled truth table for 2-fan-in gates by 25%.

Recall that in the baseline Yao’s garbled circuit, both the 0-key and 1-key for each wire are randomly chosen. After the free-XOR technique is integrated, the 0-key and 1-key for an XOR gate’s output wire depend on input key and \( R \), but the 0-key for a non-XOR gate’s output wire is still free. The GRR technique is to make a smart choice for this degree of freedom, and thus, reduce one entry in the garbled truth table to be communicated over network.

In particular, the generator picks \((Z_0, Z_1, \pi_z)\) by letting \( Z_g(0 \oplus \pi_x, 0 \oplus \pi_y) = H(X_0 \oplus \pi_x || Y_0 \oplus \pi_y) \), that is, either \( Z_0 \) or \( Z_1 \) is assigned to the encryption mask for the 0-th entry of the GTT, and the other one is computed by the equation \( Z_b = Z_1 \oplus b \oplus R \). Therefore, when the evaluator gets \((X_0 \oplus \pi_x, Y_0 \oplus \pi_y)\), both \( X_0 \oplus \pi_x \) and \( Y_0 \oplus \pi_y \) have rightmost bit 0, indicating that the 0-th entry needs to be decrypted. However, with GRR technique, she is able to retrieve \( Z_g(0 \oplus \pi_x, 0 \oplus \pi_y) \) by running \( H(\cdot) \) without inquiring GTT.

Pinkas et al. claimed that this technique is compatible with the free-XOR technique [PSSW09]. For rigorousness purposes, we carefully went through the details and came up with a security proof for our protocol that confirms this compatibility. The proof is given below in Theorem 1.

### 3.3.4 Random Seed Checking

Recall that the cut-and-choose approach requires the generator to construct multiple copies of the garbled circuit, and more than half of these garbled circuits will be fully revealed, including the randomness used to construct the circuit. Goyal, Mohassel, and Smith [GMS08] therefore pointed out an insight that the evaluator could examine the correctness of those check circuits by receiving a hash of the garbled circuit first, acquiring the random seed, and reconstructing the circuit and hash by herself.

This technique results in the communication overhead for check circuits independent of the circuit size. This technique has two phases that straddle the coin-flipping protocol. Before the coin flipping, the generator constructs multiple copies of the circuit as instructed by the cut-and-choose procedure. Then the generator sends to the evaluator the hash of each garbled circuit, rather than the circuit itself. After the coin flipping, when the evaluation circuits and the check circuits are determined, the generator sends to the evaluator the full description of the evaluation circuits and the random seed for the check circuits. The evaluator then computes the evaluation circuits and tests the check circuits by reconstructing the circuit and comparing its hash with the one received earlier. As a result, even for large circuits, the
communication cost for each check circuit is simply a hash value plus the random seed. BillionYao provides that 60% of the garbled circuits are check circuits. Thus, this optimization significantly reduces communication overhead.

3.3.5 Working with Large Circuits

A circuit for a reasonably complicated function can easily consist of billions of gates. For example, a 4095-bit edit distance circuit has 5.9 billion gates. When circuits grow to such a size, the task of achieving high performance secure computation becomes challenging.

An \((I+2C)\)-time solution Our solution for handling large circuits is based on Huang et al.’s work [HEKM11], which is the only prior work capable of handling large circuits (of up to 1.2 billion non-XOR gates) in the semi-honest setting. Intuitively, the generator could work with the evaluator in a pipeline manner so that small chunks of gates are being processed at a time. The generator could start to work on the next chunk while the evaluator is still processing the current one. However, this technique does not work directly with the random seed checking technique described above in Section 3.3.4 because the generator has to finish circuit construction and hash calculation before the coin flipping, but the evaluator could start the evaluation only after the coin flipping. As a result, the generator needs a way to construct the circuit first, wait for the coin flipping, and send the evaluation circuits to the evaluator without keeping them in memory the whole time. We therefore propose that the generator constructs the evaluation circuits all over again after the coin flipping, with the same random seed used before and the same keys for input wires gotten from OT.

We stress that when fully parallelized, the second construction of an evaluation circuit does not incur overhead to the overall execution time. Although we suggest to construct an evaluation circuit twice, the fact is that according to the random seed checking, a check circuit is already being constructed twice—once before the coin flipping by the generator for hash computation and once after by the evaluator for correctness verification. As a result, when each generator-evaluator pair is working on a single copy of the garbled circuit, the constructing time for a evaluation circuit totally overlaps with that for a check circuit. We therefore achieve the overall computation time \(I + 2C\) mentioned earlier, where the first \(C\) is for the generator to calculate the circuit hash, and the other \(C\) is either for the evaluator to reconstruct a check circuit or for both parties to work on an evaluation circuit in a pipeline manner as suggested by Huang et al. [HEKM11].

Achieving an \((I + C)\)-time solution We observe that there is a way to achieve \(I + C\) computation time, which exactly matches the running time of Yao in the semi-honest setting. This idea, however, is not compatible with the random-seed technique, and therefore represents a trade-off between communication and computation. Recall that the generator has to finish circuit construction and hash evaluation before beginning coin flipping, whereas the evaluator can start evaluating only after receiving the coin flipping results. The idea is to run the coin flipping in the way that only the evaluator gets the result and does not reveal it to the generator until the circuit construction is completed. Since the generator is oblivious to the coin flipping result, she sends every garbled circuit to the evaluator, who could then either evaluate or check the received circuit. In order for the evaluator to get the generator’s input keys for evaluation circuits and the random seed for the check circuits, they run an OT, where the evaluator uses the coin
flipping result as input and the generator provides either the random seed (for the check circuit) or his input keys (for the evaluation circuit). After the generator completes circuit construction and reveals the circuit hash, the evaluator compares the hash with her own calculation, if the hashes match, she proceeds with the rest of the original protocol. Note that this approach comes at the cost of sacrificing the random seed checking technique and its 60% savings in communication.

**Working Set Optimization**  Another problem encountered while dealing with large circuits is the *working set minimization problem*. Note that the *circuit value problem* is log-space complete for P. It is suspected that L $\neq$ P, that is, there exist some circuits that can be evaluated in polynomial time but require more than logarithmic space. This open problem captures the difficulty of handling large circuits during both the construction and evaluation, where at any moment there is a set of wires, called the *working set*, that are available and will be referenced in the future. For some circuits, the working set is inherently super-logarithmic. A naive approach is to keep the most recent $D$ wires in the working set, where $D$ is the upper bound of the input-output distance of all gates. However, there may be wires which are used as inputs to gates throughout the entire circuit, and so this technique could easily result in adding almost the whole circuit to the working set, which is especially problematic when there are hundreds of copies of a circuit of billions of gates. While reordering the circuit or adding identity gates to minimize $D$ would mitigate this problem, doing so while maintaining the topological order of the circuit is known to be an NP-complete problem, the *graph bandwidth problem* [GGJK78].

In BillionYao, the solution to this difficulty is to pre-process the circuit so that each gate comes with a usage count. The system has a compiler that converts a program in high-level language into a boolean circuit. Since the compiler is already using global optimization in order to reduce the circuit size, it is easy for the global optimizer to analyze the circuit and calculate the usage count for each gate. With this information, it is easy for the generator and evaluator to decrement the counter for each gate whenever it is being referenced and to toss away the gate whenever its counter becomes zero. In other words, we keep track of merely useful information and heuristically minimize the size of the working set, which is small compared with the original circuit size as shown in Table 3.1.

<table>
<thead>
<tr>
<th></th>
<th>AES</th>
<th>Dot(^{64})</th>
<th>RSA-32</th>
<th>EDT-255</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit size</td>
<td>49,912</td>
<td>460,018</td>
<td>1,750,787</td>
<td>15,540,196</td>
</tr>
</tbody>
</table>
| wrk set size  | 323  | 711        | 235    | 2,829

Table 3.1: The size of the working set for various circuits (sizes include input gates)

### 3.4 The Full Protocol

#### 3.4.1 Definitions

To prove the security of a garbling scheme that uses both the Free-XOR technique and the GRR3 technique, additional assumptions about the hash function $H$ are required. Choi et al. showed that rather than requiring the random oracle model it is sufficient to assume circular 2-correlation robustness [CKKZ12]. This is formalized in Definition 6, adapted from Choi et al.
The distribution of “fake” circuits will be indistinguishable from the honestly garbled circuits.

This proof follows reasoning similar to Lindell and Pinkas [LP09], as well as to Choi et al. [CKKZ12] for the reduction to circular 2-correlation robustness. At a high level, GC' will be constructed so that each garbled gate can output exactly one key, and the output wires will open only the value $z$. Any unused part of the circuit (for the input keys) will be randomly sampled placeholder values.

The circuit GC' is be constructed as follows:

1. The input keys $K'_{x}$ and $K'_{y}$ are chosen uniformly; the same key is used for every input bit value i.e. each input wire is assigned a single key.

**Definition 6** (Circular 2-Correlation Robust Hash Function [CKKZ12]). Let $\text{Circ}_R(w, w', g, b_1, b_2, b_3)$ be an oracle that outputs $H(w \oplus (b_1 R))|w' \oplus (b_1 R)||g) \oplus (b_3 R)$, and $\text{Rand}(w, w', g, b_1, b_2, b_3)$ be an oracle to a random function with the same output range. If $O$ is one of $\text{Circ}_R$ or $\text{Rand}$, let the legal queries to the oracle exclude $O(w, w', 0, 0, b_3)$, and include at most one of $O(w, w', b_1, b_2, 0)$ or $O(w, w', b_1, b_2, 1)$.

Then it is Circular 2-Correlation Robust if for any non-uniform probabilistic polynomial time distinguisher $A$ restricted to legal queries:

$$
|Pr[R \leftarrow \{0,1\}^{{\ell}_n(k)}; A^{\text{Circ}_R}(1^k) = 1] - Pr[A^{\text{Rand}}(1^k) = 1]| \leq \text{negl}(k)
$$

The BillionYao protocol builds upon the protocol of Shen and shelat [SS11], who introduced the notion of *malleable claw-free collections*. This allows for a more efficient check of the garbler’s input consistency by eliminating the need for more expensive consistency proofs involve ZKPoKs or WIPOKs. The formal definition is given in below in Definition 7.

**Definition 7** (Malleable Claw-Free Permutations [SS11]). A 4-tuple $(G_{CLW}, D_{CLW}, F_{CLW}, R_{CLW})$ is a malleable claw-free collection if the following conditions hold:

1. $(G_{CLW}, D_{CLW}, F_{CLW})$ is a claw-free collection with the range of $D_{CLW}$ and $F_{CLW}$ being groups $G_1$ and $G_2$ respectively.
2. For any $I \leftarrow G_{CLW}(1^k)$, $D_{CLW}(0, I)$ and $D_{CLW}(1, I)$ are uniformly distributed over $G_1$.
3. $R_{CLW}: G_1 \leftarrow G_2$ runs in polynomial time and for any $I, b \in \{0, 1\},$ and $m_1, m_2 \in G_1$, $f^b_I(m_1 m_2) = f^b_I(m_1) R^I_{CLW}(m_2)$.

### 3.4.2 Protocol Description

Lemma 1 is necessary to prove the security of the protocol in Figure 3.1. The process described in this lemma will be used by the simulator to construct garbled circuits that evaluate to a specific output value. The distribution of “fake” circuits will be indistinguishable from the honestly garbled circuits.

**Lemma 1** (Adapted from Claim 8 from Lindell and Pinkas [LP07]). Given a boolean circuit $C$ computing $f$ and output $z$, there exists an algorithm to compute a garbled circuit $GC'$ such that:

1. For all inputs, GC' will output $z$

2. If $z = f(x, y)$ for some $x, y$, then no non-uniform probabilistic polynomial time adversary can distinguish between the ensemble consisting of $\{GC', K'_{x'}, K'_{y'}\}$, where $K'_{x'}, K'_{y'}$ are the garbled keys for arbitrary $x', y'$, and the ensemble consisting of $\{GC, K_{x}, K_{y}\}$ where GC is garbled according to the algorithm described in Algorithm 2 and $K_{x}, K_{y}$ are the input keys for $x, y$.

**Proof.** This proof follows reasoning similar to Lindell and Pinkas [LP09], as well as to Choi et al. [CKKZ12] for the reduction to circular 2-correlation robustness. At a high level, GC' will be constructed so that each garbled gate can output exactly one key, and the output wires will open only the value $z$. Any unused part of the circuit (for the input keys) will be randomly sampled placeholder values.
1. $\mathcal{E}$ samples $I \in G_{\text{CLW}}(1^k)$ and sends $I$ to $\mathcal{G}$.

2. $\mathcal{G}$ chooses input wire keys for $\mathcal{E}$, $(K_{E,i,j}^0, K_{E,i,j}^1)_{1 \leq i \leq s, 1 \leq j \leq sn}$ uniformly. For the input wire keys for $\mathcal{G}$, $J_{E,j}^b \leftarrow H(F_{\text{CLW}}(b, I, m_{i,j}^b))$, where $m_{i,j}^b \leftarrow D_{\text{CLW}}(b, I)$, for $b \in \{0, 1\}$, $1 \leq i \leq s$, $1 \leq j \leq 2n$.

3. $\mathcal{G}$ and $\mathcal{E}$ use $s$ OT protocol instances to send the input keys $(K_{E,i,j}^b)_{1 \leq i \leq s, 1 \leq j \leq sn}$ to $\mathcal{E}$.

4. $\mathcal{G}$ sends commitments $\text{Com}(F_{\text{CLW}}(b, I, m_{i,j}^b))$ to $\mathcal{E}$ for $b \in \{0, 1\}$, $1 \leq i \leq s$, $1 \leq j \leq 2n$.

5. $\mathcal{G}$ computes $s$ copies of the garbled circuit $\{GC_i\}_{1 \leq i \leq s}$, and sends $H(GC_i)$ for each $GC_i$. The garbled circuit is modified to XOR collections of $s$ input bits for $\mathcal{E}$ for each input bit in $f$.

6. $\mathcal{G}$ and $\mathcal{E}$ run a coin-flipping protocol $\mathcal{CF}$ to select a bit string $\text{chk} \in \{0, 1\}^s$. For each $i$ where $\text{chk}_i = 0$, $\mathcal{G}$ sends the seed $\text{seed}_i$ used to generate $GC_i$. For all other $i$, $\mathcal{G}$ sends $GC_i$ and the corresponding openings to the input wire keys $J_{E,i,j}^{x_j}$.

7. $\mathcal{G}$ proves consistency with the remaining committed values. Let $e$ be the number of evaluation circuits, and let $i_1, \ldots, i_e$ be the indices of the evaluation circuits. $\mathcal{G}$ sends $m_{i_2,j}^{x_j}(m_{i_3,j}^{x_j})^{-1}, \ldots, m_{i_e,j}^{x_j}(m_{i_1,j}^{x_j})^{-1}$ to $\mathcal{E}$. $\mathcal{E}$ then checks $F_{\text{CLW}}(x_j, I, m_{i_2,j}^{x_j}) = F_{\text{CLW}}(x_j, I, m_{i_3,j}^{x_j})R_{\text{CLW}}(m_{i_2,j}^{x_j}(m_{i_3,j}^{x_j})^{-1}), \ldots, F_{\text{CLW}}(x_j, I, m_{i_e,j}^{x_j}) = F_{\text{CLW}}(x_j, I, m_{i_1,j}^{x_j})R_{\text{CLW}}(m_{i_2,j}^{x_j}(m_{i_1,j}^{x_j})^{-1})$. If the checks all pass, $J_{i,j} \leftarrow H(F_{\text{CLW}}(x_j, I, m_{i,j}^{x_j}))$.

8. $\mathcal{E}$ checks for consistency of each garbled circuit $GC_i$ with the hashes and evaluates the circuits to get garbled outputs $W_{\text{E},i,j}^0$ and sends commitments to the output wire values $\text{Com}(W_{\text{E},i,j}^0)$. If a circuit cannot be evaluated, a commitment to a random value is sent.

9. $\mathcal{G}$ opens the output wire values $W_{\text{E},i,j}^0, W_{\text{E},i,j}^1$.

10. $\mathcal{E}$ checks for consistency between $W_{\text{E},i,j}^0$ and $W_{\text{E},i,j}^0, W_{\text{E},i,j}^1$ and selects the majority output value for each output bit $b_j$. $\mathcal{E}$ uses a witness-indistinguishable proof to prove that $b_j$ is consistent with the committed $W_{\text{E},i,j}$ values.

11. $\mathcal{G}$ unmask its output value.

Figure 3.1: The full BillionYao protocol. The evaluator is denoted $\mathcal{E}$, and the generator is denoted $\mathcal{G}$. $H$ is sampled from a family of collision resistant hash functions, and $G_{\text{CLW}}$ is a family of malleable claw-free permutations.
Algorithm 2 Garbling with the Free-XOR and Garbled Row Reduction (GRR3) optimizations.

Input: Circuit \( C \) consisting only of AND and XOR gates, input wire keys \( \left( J_{g,i}^0, J_{g,i}^1 \right) \) \( 1 \leq j \leq 2^n \) and \( \left( K_{E,j}^0, K_{E,j}^1 \right) \) \( 1 \leq j \leq m \).

Output: Garbled Circuit \( GC \)

Sample \( R' \leftarrow \{0, 1\}^{k-1} \) and set \( R \leftarrow R'||1 \).

For each AND gate \( g_i \) with input wires \( L, R \), let \( b_L, b_R \) be the bits generated for the left and right input wires. Set \( W_i^{b_L, b_R} \leftarrow H(W_i^L || W_i^R || i) \), and set \( b_i \leftarrow \text{LOB}(W_i^{b_L \land b_R} \oplus (b_L \land b_R) \land W_i^{1 \oplus (b_L \land b_R)}) \leftarrow W_i^{b_L \land b_R} \oplus R \). Compute the garbled gate \( GC_i \) as:

\[
GC_i \leftarrow (H(W_i^L || W_i^R || g_i) \oplus W_i^{(1 \oplus b_L) \land b_R},
H(W_i^L || W_i^{1 \oplus b_R} || g_i) \oplus W_i^{b_L \land (1 \oplus b_R)},
H(W_i^L \oplus b_L || W_i^R || g_i) \oplus W_i^{(1 \oplus b_L) \land (1 \oplus b_R)})
\]

For each output wire \( \text{Out}_{C;i} \in C \), sample \( W_{\text{Out;i}}^0, W_{\text{Out;i}}^1 \leftarrow \{0, 1\}^k \) and \( b \leftarrow \{0, 1\} \) and set \( GC_{\text{Out;i}} \leftarrow (H(W_0^i || i) \oplus W_0^i || H(W_1^i || i) \oplus W_1^i \land \text{Com}(W_{\text{Out;i}}^0), \text{Com}(W_{\text{Out;i}}^1)) \)

For each output wire \( \text{Out}_{E,j} \in C \), set \( GC_{\text{Out;j}} \leftarrow (H(W_0^j), H(W_1^j)) \). These are sent in order and will be used to decode the garbled output.

2. For each non-XOR gate \( i \), there are four cases:
   
   - \( b_L = 0, b_R = 0 \) Set \( W_i \leftarrow H(W_L || W_R || i) \) and sample \( GC_i \leftarrow \{GC_i^1, GC_i^2, GC_i^3\} \) uniformly; \( GC_i \leftarrow \{H(W_L || W_R || g_i) \oplus W_i, GC_i^1\} \)
   - \( b_L = 1, b_R = 0 \) \( W_i \) is sampled uniformly, and sample \( GC_i \leftarrow \{H(W_L || W_R || g_i) \oplus W_i, GC_i^2\} \)
   - \( b_L = 0, b_R = 1 \) \( W_i \) is sampled uniformly, and sample \( GC_i \leftarrow \{H(W_L || W_R || g_i) \oplus W_i, GC_i^3\} \)
   - \( b_L = 1, b_R = 1 \) \( W_i \) is sampled uniformly, and sample \( GC_i \leftarrow \{GC_i^1, H(W_L || W_R || g_i) \oplus W_i, GC_i^1\} \)

   Set \( b_i \leftarrow \text{LOB}(W_i) \).

3. For each XOR gate \( k \), set \( W_k \leftarrow W_L \oplus W_R \).

4. For each output wire \( \text{Out}_{E,j} \), set \( GC_{E,j} \leftarrow \{H(W_{E,j}), \text{Random}\} \) if \( z_j = 0 \), or \( GC_{E,j} \leftarrow \{\text{Random}, H(W_{E,j})\} \) otherwise.

Note that for each wire there is exactly one key, \( W_i \), and that the wire key used for the evaluator output wires can open exactly the value for \( z \) regardless of the input values \( x', y' \). The argument that \( GC' \) is indistinguishable from the honestly constructed \( GC \) proceeds by a hybrid proof on the non-XOR gates and a reduction to the circular 2-correlation robustness of \( H \).

Let \( w_i \) be the bit value of each output wire of gate \( i \) in the (non-garbled) circuit when it is evaluated on \( f(x, y) \). Let \( GC \) be an honestly generated garbled circuit with input wire keys \( K_x, K_y \) corresponding to the inputs \( x, y \). We construct hybrids as follows:

For hybrid \( i \), replace the rows in the honestly garbled gate \( i \) with oracle queries to \( \mathcal{O}(W_L^{w_L}, W_R^{w_R}, i, \ldots) \oplus W_i^{w_i} \) from Definition 6, except for the \( H(W_L^{b_L \oplus w_L} || W_R^{b_R \oplus w_R} || i) \) row if \( b_L \neq 0 \land b_R \neq 0 \). If \( w_L \land w_R = w_i \), set \( b_3 = 0 \) in the oracle query; otherwise, \( b_3 = 1 \).
In the final hybrid, the output wire values are replaced; indistinguishibility follows from the collision resistance of $H$.

If $O$ is the Circ$^R$ oracle, this will be the honestly garbled gate. Otherwise, when $O$ is Rand, this is the GC$'_i$ gate described above. This completes the reduction to the circular 2-correlation robustness of $H$. 

Lemma 1 will be used by the simulator in Theorem 1 below. The proof of Theorem 1 is similar to the proof from shelat and Shen [SS11], but is modified as needed for the different techniques used in the BillionYao protocol.

Theorem 1. The protocol described in Figure 3.1 securely computes $f$ with aborts in the presence of static malicious adversaries.

Proof. First, consider the case of a malicious evaluator $E^*$.

Claim 1. REAL($G(x), E^*(y), 1^k$) $\approx$ Ideal($G(x), SIM^{E^*}(y)$)

Proof. The proof proceeds by a hybrid argument, beginning with Hyb$_0$, which replaces the real OTs with EXT$^{OR}$s. Note that EXT$^{OR}$ extracts each bit $y_i^*$ of $E^*(y)$’s inputs. Hyb$_0$ is indistinguishable from REAL from the security of the OT protocol. At this point, SIM$^{E^*}(y)$ has extracted the inputs, and receives from the trusted party $f_E(x, y)$.

The next hybrid, Hyb$_1$, replaces the coin-flipping protocol with SIM$^{CF}($ for some randomly sampled string $\rho \in \{0, 1\}^*$ with the correct Hamming weight. Indistinguishability follows from the security of the coin-flipping protocol $CF$.

Hyb$_3$ is the same as Hyb$_1$ except that it replaces the garbled circuits corresponding to $b_i = 1$ with a circuit that outputs $f_E(x, y)$ using the process from Lemma 1. For $b_i = 0$, corresponding to the check circuits, the circuit is garbled honestly. Indistinguishability follows from Lemma 1. Note that at this point, the input keys $J_{G, i,j}$ do not depend on $x$, as the construction in Lemma 1 assigns the same key to both 0 and 1 for each input bit.

Hyb$_4$ replaces each of $G$’s input bits with 0 in the input consistency check. In particular, rather than open $F_{CLW}(x_j, I, m^{x_j})$, in this hybrid $E$ will receive openings to $F_{CLW}(0, I, m^{x_j}_0)$. Consistency will be proved by sending $m^{0}_{i,j}(m^{0}_{i,j})^{-1}, \ldots, m^{0}_{i,j}(m^{0}_{i,j})^{-1}$. Indistinguishability follows immediately from the fact that $D_{CLW}(b, I)$ follows a uniform distribution. Note that at this point $G$’s input is not needed.

Finally, Hyb$_4$ is identical to Hyb$_3$ except that it sends Abort to the ideal party if the WIP fails. Indistinguishability follows from the soundness of the witness-indistinguishable proof. This completes the proof, with Hyb$_4$ being the simulator.

Claim 2. SIM$^{E^*}(y)$ runs in expected polynomial time.

Proof. Both Hyb$_0$ and Hyb$_1$, run in expected polynomial time, from the security of OT and CF respectively. The remaining hybrids do not require any rewinding and run in polynomial time.

Claim 3. REAL($G^*(x), E(y), 1^k$) $\approx$ Ideal($SIM^{G^*}(x), E(y)$)

Proof. The proof proceeds by a hybrid argument.

Hyb$_0$ differs from REAL($G^*(x), E(y), 1^k$) by rewinding $G^*$ if the cut-and-choose check passes, repeating the experiment with different randomness for $CF$ until it passes again. If there is no index $i$ in which
the cut-and-choose check circuits differ between the two passing experiments, the simulator will abort. Indistinguishability follows from the same argument given by Shelat and Shen [SS11].

Hyb₁ extracts the input from the garbled circuit at position $i$ from Hyb₀, or aborts if the input cannot be properly extracted. There are two possible failures: $fail_1$ occurs if the opened commitment in the case where $GC_i$ is an evaluation circuit does not match the values from the check circuit case; $fail_2$ occurs if for two circuits the extracted inputs are not the same. $fail_1$ cannot happen because of the binding properties of the commitment scheme. $fail_2$ requires $G^*$ to find a claw in the malleable claw-free permutation, which happens with vanishingly small probability. Thus $Hyb_1 \approx Hyb_0$.

Hyb₂ sends the extracted input $x$ to the trusted party and receives output $f_G(x, y)$, aborting if the output is inconsistent with the majority output. This abort will occur only if $G$ successfully corrupts a majority of the evaluation circuits, which happens with vanishingly small probability.

Hyb₃ uses the value received from the trusted party in the output authenticity check. Picking the circuit at index $i$ from Hyb₀, the output wire keys corresponding to $f_G(x, y)$ are committed to, and the knowledge of both output wire keys is used to complete the WIP. If the circuit evaluation fails, i.e. if there is no majority output, Hyb₃ aborts.

Hyb₄ does not evaluate the garbled circuits at all, and uses only the extracted input value and the trusted party’s output. In other words, Hyb₄ cannot fail due to the circuit evaluation failing, so it suffices to prove that this happens with negligible probability to prove $Hyb_4 \approx Hyb_3$. This follows by the same reasoning from Shelat and Shen [SS11].

Hyb₅ uses random inputs in the OT protocol. The security of the OT protocol ensures that the two hybrids are indistinguishable during the OT phase, it remains to be proved that indistinguishibility holds after the OTs. It is possible that in Hyb₄ one of the keys $K_{1-y}^{i,j}$ is corrupt and that the circuit $GC_i$ will not be evaluable, while $K_{1-y}^{i,j}$ is not corrupt; if the corrupt key is chosen it will be detected when the check circuits are revealed. If $GC_j$ is an evaluation circuit, the corruption will never be detected, since Hyb₄ does not evaluate any circuits.

Due to the random input replacement technique, the distribution of inputs between Hyb₄ and Hyb₅ are statistically close; thus $Hyb_4 \approx Hyb_3$. This completes the proof, with Hyb₅ as the simulator.

**Claim 4.** $SIM^{G^*(x)}$ runs in expected polynomial time.

**Proof.** Hyb₀ must rewinding until the cut-and-choose phase has passed twice. Since each attempt will require polynomial time, it follows from the bound on $G^*$’s success probability that Hyb₀ runs in expected polynomial time. This argument is identical to the argument given by Shelat and Shen [SS11].

The remaining hybrids do not require any rewinding, beyond the rewinding done by the simulators for subprotocols, and do only polynomial time operations.

**3.5 Boolean Circuit Compiler**

Although the Fairplay circuit compiler can generate circuits, it requires a very large amount of computational resources to generate even relatively small circuits. Even on a machine with 48 gigabytes of RAM, Fairplay terminates with an out-of-memory error after spending 20 minutes attempting to compile an AES circuit. This makes Fairplay impractical for even relatively small circuits, and infeasible for some
of the circuits tested in this project. One goal of this project was to have a general purpose system for secure computation, and so writing application specific programs to generate circuits, a technique used by others [HEKM11], was not an option.

To address this problem, BillionYao includes a compiler that generates a more efficient output format than Fairplay, and which requires far lower computational resources to compile circuits. This compiler is able to generate the AES circuit in only a few seconds on a typical desktop computer with only 8GB of RAM, and were able to generate and test much larger non-trivial circuits. The compiler is implemented using the well-known flex and bison tools to generate our compiler, and implemented an optimizer as a separate tool. As an optimization it uses the results from [PSSW09] to reduce 3 arity gates to 2 arity gates with minimal non-XORs.

As a design decision, BillionYao uses a imperative, untyped language with static scoping. The language allows code, variables, and input/output statements to exist in the global scope; this allows very simple programs to be written without too much extra syntax. Functions may be declared, but may not be recursive. Variables do not need to be declared before being used in an unconditional assignment; variables assigned within a function’s body that are not declared in the global scope are considered to be local. Arrays are a language feature, but array indices must be constants or must be determined at compile time. If run-time determined indices are required for a function, a loop that selects the correct index may be used; this is necessary for oblivious evaluation. Variables may be arbitrarily concatenated, and bits or groups of bits may be selected from any variable and bits or ranges of bits may be assigned to; as with arrays, the index of a bit must be determined at compile time, or else a loop must be used. Note that loop variables may be used as such an index, since loops are always completely unrolled, and therefore the loop index can always be resolved at compile time. Additional language features are planned as future work.

Some techniques from the Fairplay compiler are also used in the BillionYao compiler. In particular it uses the single assignment algorithm from Fairplay, which is required to deal with assignments that occur inside of if statements. Otherwise, the compiler has several distinguishing characteristics that make it more resource efficient than Fairplay. The front end attempts to generate circuits as quickly as possible, using as little memory as possible and performing only rudimentary optimizations before emitting its output. This can be done with very modest computational resources, and the intermediate output can easily be translated into a circuit for evaluation. The main optimizations are performed by the back end of the compiler, which identifies gates that can be removed without affecting the output of the circuit as a whole.

Unlike the Fairplay compiler, BillionYao avoids the use of hash tables in the compiler, instead using more memory-efficient storage. The system can use one of three storage strategies: memory-mapped files, flat files without any mapping, and Berkeley DB. As presented below, memory mapped files always resulted in the highest performance, but that Berkeley DB is only sometimes better than direct access without any mapping.

3.5.1 Circuit Optimizations

The compiler front-end tends to generate inefficient circuits, with large numbers of unnecessary gates. As an example, for some operations the compiler generates large numbers of identity gates i.e. gates whose outputs follow one of their inputs. It is therefore essential to optimize the circuits emitted by the front end, particularly to meet BillionYao’s overall goal of practicality.
Figure 3.2: Average fraction of circuits remaining after each optimization is applied in sequence. Note that the *relative change* in circuit sizes after each optimization is dependent on the circuit itself, with some circuits being optimized more than others.

The compiler uses several stages of optimization, most of which are global. As a first step, a local optimization removes redundant gates, i.e. gates that have the same truth table and input wires. This first step operates on a fixed-size chunk of the circuit, but tests revealed that there are diminishing improvements as the size of this window is increased. It also removes constant gates, identity gates, and inverters, which are generated by the compiler and which may be inadvertently generated during the optimization process. Finally, it removes gates that do not influence the output, analogous to dead code elimination. The effectiveness of each optimization on different circuits is shown in Figure 3.2. The circuit that was least optimizable was the edit distance circuit, being reduced to only 82% of its size from the front end, whereas the RSA signing and the dot product circuits were the most optimizable, being reduced to roughly half of the gates emitted by the front end.

**Gate Removal** The front-end of the compiler emits gates in topological order, and similar to Fairplay, the compiler assigns explicit identifiers to each emitted gate. To remove gates efficiently, it stores a table that maps the identifiers of gates that were removed to the previously emitted gates, and for each gate that is scanned the inputs are rewritten according to this table. The table itself is then emitted, so that the identifiers of non-removed gates can be corrected. This mapping process can be done in linear time and space using an appropriate key-value store.

**Removing Redundant Gates** Some of the gates generated by the front-end have the same truth table and input wires as previously generated gates; such gates are redundant and can be removed. This removal process has the highest memory requirement of any other optimization step, since a description of every non-redundant gate must be stored. However, experimental results revealed that this optimization can be performed on discrete chunks of the circuit with results that are very close to performing the optimization on the full circuit, and that there are diminishing improvements in effectiveness as the size of the chunks is increased. Therefore, this optimization is performed using chunks, and it is possible to use hash tables to improve the speed of this step.
Removing Identity Gates and Inverters  The front end may generate identity gates or inverters, which are not necessary. This may happen inadvertently, such as when a variable is incremented by a constant, or as part of the generation of a particular logic expression. While removing identity gates is straightforward, the removal of inverters requires more work, as gates which have inverted input wires must have their truth tables rewritten. There is a cascading effect in this process; the removal of some identity gates or inverters may transform later gates into identity gates or inverters. This step also removes gates with constant outputs, such as an XOR gate with two identical inputs. Constant propagation and folding occur as a side effect of this optimization.

Removing Unused Gates  Finally, some gates in the circuit may not affect the output value at all. For this step, the optimizer will scan the circuit backwards, and store a table of live gates; it will then re-emit the live gates in the circuit and skip the dead gates. Immediately following this step, the circuit is prepared for the garbled circuit generator, which includes generating a usage count for each gate.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>DB (s)</th>
<th>mmap (s)</th>
<th>flat (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7200RPM Spinning Disk (ext4-fs)</td>
<td>4.3 ±0.5%</td>
<td>1.05 ± 1%</td>
<td>3.48 ±0.3%</td>
</tr>
<tr>
<td>AES</td>
<td>103 ±0.3%</td>
<td>24.6 ±0.2%</td>
<td>78.4 ±0.3%</td>
</tr>
<tr>
<td>RSA-32</td>
<td>32.56 ±0.1%</td>
<td>7.1 ±0.3%</td>
<td>28.37 ±0.1%</td>
</tr>
<tr>
<td>Dot\textsuperscript{64}\textsubscript{4}</td>
<td>975 ±0.1%</td>
<td>240 ± 1%</td>
<td>700 ±0.9%</td>
</tr>
<tr>
<td>EDT-255</td>
<td>3.62 ±0.3%</td>
<td>0.86 ± 1%</td>
<td>3.17 ±0.6%</td>
</tr>
<tr>
<td>Solid-State Drive</td>
<td>96.5 ±0.2%</td>
<td>21.6 ±0.4%</td>
<td>68.3 ±0.3%</td>
</tr>
<tr>
<td>AES</td>
<td>30.5 ±0.5%</td>
<td>6.27 ± 1%</td>
<td>25.9 ±0.2%</td>
</tr>
<tr>
<td>RSA-32</td>
<td>907 ±0.1%</td>
<td>200 ±0.4%</td>
<td>590 ± 1%</td>
</tr>
<tr>
<td>Dot\textsuperscript{64}\textsubscript{4}</td>
<td>5.56 ± 4%</td>
<td>1.12 ± 0%</td>
<td>7.11 ±0.3%</td>
</tr>
<tr>
<td>EDT-255</td>
<td>208 ±0.4%</td>
<td>45.7 ± 3%</td>
<td>240 ±0.1%</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>46.3 ±0.1%</td>
<td>9.2 ±0.2%</td>
<td>60.7 ±0.2%</td>
</tr>
<tr>
<td>AES</td>
<td>2500 ± 1%</td>
<td>405 ±0.2%</td>
<td>2050 ±0.2%</td>
</tr>
</tbody>
</table>

Table 3.2: Compile times for different storage systems for small circuits (sizes include input gates), using different storage media. Results are averaged over 30 experiments, with 95% confidence intervals. On EC2, a high-memory quadruple extra large instance was used.

Key-Value Stores  Unfortunately, even though the compiler is more resource efficient than Fairplay, it still requires space that is linear in the size of the circuit. For very large circuits, circuits with billions of gates or more, this may exceed the amount of RAM that is available. The compiler can make use of a computer’s hard drive to store intermediate representations of circuits and information about how
to remove gates from the circuit. The implementation used memory-mapped I/O to reduce the impact this has on performance; however, the use of `mmap` and `ftruncate` is not portable, and so BillionYao also supports using an unmapped file or Berkeley DB. Tests revealed that, as expected, memory-mapped I/O achieves the highest performance, but that Berkeley DB is sometimes better than unmapped files on high-latency filesystems. A summary of the performance of each method on a variety of storage systems is shown in Table 3.2.

Using the hard drive in this manner is crucial for compiling the largest circuits presented in the experiments. The performance impact of writing to disk should not be understated; a several-billion-gate edit distance 4095x4095 circuit required more than 3 days to compile on an Amazon EC2 high-memory image, with 68 GB of RAM, one third of which was spent waiting on I/O. This, however, is a one-time cost; a compiled circuit can be used in unlimited evaluations of a secure computation protocol.

<table>
<thead>
<tr>
<th>RSA Size</th>
<th>Circuit Size</th>
<th>Compile Time (s)</th>
<th>Gates/s</th>
<th>Edit-Dist Size</th>
<th>Circuit Size</th>
<th>Compile Time (s)</th>
<th>Gates/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>208,499</td>
<td>2.6 ± 7%</td>
<td>80,000</td>
<td>31x31</td>
<td>144,277</td>
<td>1.70 ±0.7%</td>
<td>84,900</td>
</tr>
<tr>
<td>32</td>
<td>1,750,787</td>
<td>21.6 ±0.4%</td>
<td>81,100</td>
<td>63x63</td>
<td>717,233</td>
<td>8.56 ±0.7%</td>
<td>83,800</td>
</tr>
<tr>
<td>64</td>
<td>14,341,667</td>
<td>189 ±0.3%</td>
<td>75,900</td>
<td>127x127</td>
<td>3,389,812</td>
<td>41.7 ±0.5%</td>
<td>81,300</td>
</tr>
<tr>
<td>128</td>
<td>116,083,983</td>
<td>1810 ±0.3%</td>
<td>64,100</td>
<td>255x255</td>
<td>15,540,196</td>
<td>200 ±0.4%</td>
<td>77,700</td>
</tr>
</tbody>
</table>

Table 3.3: Time required to compile and optimize RSA and edit distance circuits on a workstation with an Intel Xeon 5506 CPU, 8GB of RAM and a 160GB SSD, using the textbook modular exponentiation algorithm. Note that the throughput for edit distance is higher even for comparably sized circuits; this is because the front end generates a more efficient circuit without any optimization. Compile times are averaged over 30 experiments, with 95% confidence intervals reported.

### 3.5.2 Compiler Testing Methodology

Five circuits were used to test the compiler’s performance. The first was AES, as a comparison with the Fairplay system. AES with the compact S-Box description given by Boyar and Parelta [BP10] was also tested, which results in a smaller AES circuit. An RSA signing circuit with various toy key sizes, up to 128 bits, was used as a test of the compiler’s handling of large circuits; RSA circuits have cubic size complexity, allowing us to generate very large circuits with small inputs. An edit distance circuit was also tested to compare BillionYao with the work of Huang et al. [HEKM11]; unlike the other test circuits, there is no multiplication routine in the inner loop of this function. The final test circuit was a dot product with error, a basic sampling function for the LWE problem, which is similar to RSA in creating large circuits, but also demonstrates BillionYao’s ability to handle large input sizes.

After compiling these circuits, the correctness of the garbled circuit implementation was tested by first performing a direct, offline evaluation of the circuit, and comparing the output to a non-circuit implementation. Next the output of an online evaluation was checked against the offline evaluation. Additionally, for the AES circuit, the output of the circuit generated by the compiler was check against the output of a circuit generated using Fairplay. All three key-value stores were tested on a variety of file systems, including a fast SSD, a spinning disk, and an Amazon EC2 instance store, checking for correctness as described above in each case.
3.5.3 Summary of Compiler Performance

The compiler is able to emit and optimize large circuits in relatively short periods of time, less than an hour for circuits with tens of millions of gates on an inexpensive workstation. Figure 3.2 summarizes the effectiveness of the various optimization stages on different circuits; in circuits that involve multiplication in finite fields or modulo an integer, the identity gate removal step is the most important, removing more than half of the gates emitted by the front-end. The edit distance circuit is the best-case for the front-end, as less than 1/5 of the gates that are emitted can be removed by the optimizer. The throughput of the compiler is dependent on the circuit being compiled, with circuits which are more efficiently generated by the front-end being compiled faster; in Table 3.3 a comparison the generation of RSA circuits to edit distance circuits is presented.

3.6 Experimental Results

The experimental environment was the Ranger cluster in the Texas Advanced Computing Center. Ranger is a blade-based system, where each node is a SunBlade x6240 blade running a Linux kernel and has four AMD Opteron quad-core 64-bit processors, as an SMP unit. Each node in the Ranger system has 2.3 GHz core frequency and 32 GB of memory, and the point-to-point bandwidth is 1 GB/sec. Although Ranger is a high-end machine, the experiments used only a small fraction of its power, only 512 out of 62,976 cores. Note that the PBC (Pairing-Based Cryptography) library [Lyn06] to implement the underlying cryptographic protocols such as oblivious transfers, witness-indistinguishable proofs, and so forth. However, moving to more modern libraries such as RELIC [Rel] is likely to give even better results, especially to those circuits with large input and output size.

System Setup  In BillionYao, both the generator and the evaluator run an equal number of processes, including a root process and many slave processes. The root process is responsible for coordinating its own slave processes and the other root process, while the slave processes work together on repeated and independent tasks. There are three pieces of code in BillionYao: the generator, the evaluator, and the IP exchanger. Both the generator’s and evaluator’s program are implemented with Message Passing Interface (MPI) library. The reason for the IP exchanger is that it is common to run jobs on a cluster with dynamic working node assignment. However, when the nodes are dynamically assigned, the generator running on one cluster and the evaluator running on another might have a hard time locating each other. Therefore, a fixed location IP exchanger helps the match-up process as described in Figure 3.3. BillionYao has two modes—the user mode and the simulation mode. The former works as mentioned above, and the latter simply spawns an even number of processes, half for the generator and the other half for the evaluator. The network match-up process is omitted in the latter mode to simplify the testing of this system.

To achieve a security level of $2^{-80}$, meaning that a malicious player cannot successfully cheat with probability better than $2^{-80}$, requires at least 250 copies of the garbled circuit [SS11]. For simplicity, the experimental setup used 256 copies, that is, the security parameters were $k = 80$ and $s = 256$. Each experiment was run 30 times (unless stated otherwise), and the following sections report the average runtime of the experiments.
Timing methodology When there is more than one process on each side, care must be taken in measuring the timings of the system. The timings reported in this section are the time required by the root process at each stage of the system. This was chosen because the root process will always be the longest running process, as it must wait for each slave process to run to completion. Moreover, in addition to doing all the work that the slaves do, the root processes also perform the input consistency check and the coin tossing protocol.

Impacts of the Performance Optimization Techniques Several performance optimization techniques were described above in Section 3.3 with theoretical analyses, and the experiments in this section demonstrate their empirical effectiveness in Table 3.4. As expected, the Random Seed Checking technique reduces the communication cost for the garbled circuits by 60%, and the Garbled Row Reduction further reduces by another 25%. In the RS and GRR columns, the small deviation from the theoretical fraction 40% and 30%, respectively, is due to certain implementation needs. The compiler is designed to reduce the number of non-XOR gates. In these four circuits, the ratio of non-XOR gates is less than 43%. So after further applying the Free-XOR technique, the final communication is less than 13% of that in the baseline approach.

<table>
<thead>
<tr>
<th></th>
<th>non-XOR (%)</th>
<th>Baseline (MB)</th>
<th>RS (%)</th>
<th>GRR (%)</th>
<th>FX (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>30.81</td>
<td>509</td>
<td>39.97</td>
<td>30.03</td>
<td>9.09</td>
</tr>
<tr>
<td>Dot_64</td>
<td>29.55</td>
<td>4,707</td>
<td>39.86</td>
<td>29.91</td>
<td>8.88</td>
</tr>
<tr>
<td>RSA-32</td>
<td>34.44</td>
<td>17,928</td>
<td>39.84</td>
<td>29.88</td>
<td>10.29</td>
</tr>
<tr>
<td>EDT-255</td>
<td>41.36</td>
<td>159,129</td>
<td>39.84</td>
<td>29.87</td>
<td>12.36</td>
</tr>
</tbody>
</table>

Table 3.4: The impact of various optimization techniques: The Baseline shows the communication cost for 256 copies of the original Yao garbled circuit when \( k = 80 \); RS shows the remaining fraction after Random Seed technique is applied; GRR shows when Garbled Row Reduction is further applied; and FX shows when the previous two techniques and the Free-XOR are applied. (The communication costs here only include those in the generation and evaluation stages.)

Performance Gain by AES-NI On a machine with 2.53 GHz Intel Core i5 processor and 4GB 1067 MHz DDR3 memory, it took 784 clock cycles to run a single SHA-256 (with OpenSSL 1.0.0g), while it
needs only 225 cycles for AES-256 (with AES-NI). To measure the benefits of AES-NI, two instantiations were used to construct various circuits, listed in Table 3.5; observe the consistent 20% saving in circuit construction.\(^3\)

<table>
<thead>
<tr>
<th>Type</th>
<th>size (gate)</th>
<th>AES-NI (sec)</th>
<th>SHA-256 (sec)</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>49,912</td>
<td>0.12±1%</td>
<td>0.15±1%</td>
<td>78.04</td>
</tr>
<tr>
<td>Dot(^4)</td>
<td>460,018</td>
<td>1.11±0.4%</td>
<td>1.41±0.5%</td>
<td>78.58</td>
</tr>
<tr>
<td>RSA-32</td>
<td>1,750,787</td>
<td>4.53±0.5%</td>
<td>5.9±0.8%</td>
<td>76.78</td>
</tr>
<tr>
<td>EDT-255</td>
<td>15,540,196</td>
<td>42.0±0.5%</td>
<td>57.6±1%</td>
<td>72.92</td>
</tr>
</tbody>
</table>

Table 3.5: Circuit generation time (for a single copy) with different instantiations (AES-NI vs SHA-256) of the 2-circular correlation robust function.

**AES** An AES circuit was used as a benchmark to compare our compiler to the Fairplay compiler, and as a test circuit for our system. This test used full AES circuit, as specified in FIPS-197 [FIP01]. In the semi-honest model, it is possible to reduce the number of gates in an AES circuit by computing the key schedule offline; e.g. this is one of the optimizations employed by Huang et al. [HEKM11]. In the malicious model, however, such an optimization is not possible; the key-holding party could compute the key schedule maliciously. Therefore in these experiments the entire function, including the key schedule, was computed online.

In these experiments, the two parties collaboratively compute the function \(f : (x, y) \mapsto (\perp, \text{AES}_x(y))\), i.e. the circuit generator holds the encryption key \(x\), while the evaluator has the message \(y\) to be encrypted. At the end, the generator will not receive any output, whereas the evaluator will receive the ciphertext \(\text{AES}_x(y)\).

<table>
<thead>
<tr>
<th>Type</th>
<th>Fairplay</th>
<th>Ours-A</th>
<th>Pinkas et al.</th>
<th>Ours-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-XOR</td>
<td>15,316</td>
<td>15,300</td>
<td>11,286</td>
<td>9,100</td>
</tr>
<tr>
<td>XOR</td>
<td>35,084</td>
<td>34,228</td>
<td>22,594</td>
<td>21,628</td>
</tr>
</tbody>
</table>

Table 3.6: The components of the AES circuits from different sources. Ours-A comes from the textbook AES algorithm, and Ours-B uses an optimized S-box circuit from [BP10]. (Sizes do not include input or output wires)

First, note the performance of the compiler in Table 3.6. As described in Section 3.5 that the compiler is capable of large circuit generation. The experiments also reveal that the compiler produces a smaller circuit than Fairplay for this function.

Given the same high-level description of AES encryption (textbook AES), the BillionYao compiler produces a circuit with a smaller gate count and even fewer non-XOR gates. When applying the compact S-Box description proposed by Boyar and Parelta [BP10] to the high-level description as input, a smaller AES circuit than the hand-optimized one from Pinkas et al. is generated, with less programmer effort.

In Table 3.7, both the computational and communication costs for each main stage are listed under the traditional setting, where there is only one process on each side. These main stages include oblivious transfer, garbled circuit construction, the generator’s input consistency check, and the circuit evaluation.

\(^3\)The reason that saving 500+ cycles does not lead to more improvements is that this encryption operation is merely one of the contributing factors to generating a garbled gate. Other factors, for example, include GNU hash_map table insertion (~1,200 cycles) and erase (~600 cycles).
Table 3.7: The 95% two-sided confidence intervals of the computation and communication time for each stage in the experiment \((x, y) \rightarrow (\bot, \text{AES}_x(y))\).

<table>
<thead>
<tr>
<th></th>
<th>Gen (sec)</th>
<th>Eval (sec)</th>
<th>Comm (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OT comp</td>
<td>45.8±0.09%</td>
<td>34.0±0.2%</td>
<td>5.516</td>
</tr>
<tr>
<td>OT comm</td>
<td>0.1±1%</td>
<td>11.9±0.6%</td>
<td></td>
</tr>
<tr>
<td>Gen. comp</td>
<td>35.6±0.5%</td>
<td>--</td>
<td>3</td>
</tr>
<tr>
<td>Gen. comm</td>
<td>--</td>
<td>35.6±0.5%</td>
<td></td>
</tr>
<tr>
<td>Inp. comp</td>
<td>--</td>
<td>1.75±0.2%</td>
<td>266</td>
</tr>
<tr>
<td>Inp. comm</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>Evl. comp</td>
<td>14.9±0.6%</td>
<td>32.4±0.4%</td>
<td>28,781</td>
</tr>
<tr>
<td>Evl. comm</td>
<td>18.2±1%</td>
<td>3.2±0.8%</td>
<td>34,566</td>
</tr>
<tr>
<td>Total comp</td>
<td>96.3±0.3%</td>
<td>68.0±0.2%</td>
<td></td>
</tr>
<tr>
<td>Total comm</td>
<td>18.3±1%</td>
<td>50.8±0.4%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: The average and error interval of the times (seconds) running AES circuit. The number of nodes represents the degree of parallelism on each side. “–” means that the time is smaller than 0.05 seconds. Inter-com refers to the communication between the two parties, and intra-com refers to communication between nodes for a single party.

<table>
<thead>
<tr>
<th>node #</th>
<th>Gen 4</th>
<th>Evl</th>
<th>Gen 16</th>
<th>Evl</th>
<th>Gen 64</th>
<th>Evl</th>
<th>Gen 256</th>
<th>Evl</th>
</tr>
</thead>
<tbody>
<tr>
<td>OT</td>
<td>12.56±0.1%</td>
<td>8.41±0.1%</td>
<td>4.06±0.1%</td>
<td>2.13±0.2%</td>
<td>1.96±0.1%</td>
<td>0.58±0.2%</td>
<td>0.64±0.1%</td>
<td>0.19±0.2%</td>
</tr>
<tr>
<td>Gen.</td>
<td>8.18±0.4%</td>
<td>--</td>
<td>1.92±0.7%</td>
<td>--</td>
<td>0.49±0.4%</td>
<td>--</td>
<td>0.14± 1%</td>
<td>--</td>
</tr>
<tr>
<td>Inp. Chk</td>
<td>--</td>
<td>0.42± 4%</td>
<td>--</td>
<td>0.10±10%</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Evl.</td>
<td>3.3± 4%</td>
<td>7.08± 1%</td>
<td>0.80±10%</td>
<td>1.58± 4%</td>
<td>0.23±17%</td>
<td>0.37± 7%</td>
<td>0.12±0.5%</td>
<td>0.05±0.6%</td>
</tr>
<tr>
<td>Inter-com</td>
<td>4± 5%</td>
<td>13.2±0.3%</td>
<td>0.93±10%</td>
<td>4.08±0.8%</td>
<td>0.31±20%</td>
<td>1.98± 1%</td>
<td>0.11±40%</td>
<td>0.72±0.2%</td>
</tr>
<tr>
<td>Intra-com</td>
<td>0.17±30%</td>
<td>0.23±20%</td>
<td>0.18± 8%</td>
<td>0.25± 6%</td>
<td>0.45±20%</td>
<td>0.48±15%</td>
<td>0.34±30%</td>
<td>0.34±30%</td>
</tr>
<tr>
<td>Total time</td>
<td>28.3±0.3%</td>
<td>29.4±0.3%</td>
<td>7.90±0.5%</td>
<td>8.17±0.4%</td>
<td>3.45± 2%</td>
<td>3.44± 2%</td>
<td>1.4± 10%</td>
<td>1.3± 9%</td>
</tr>
</tbody>
</table>

Each row includes both the computation and communication time used. Note that network conditions could vary from setting to setting. The experiments ran in a local area network, and the data can only give a rough idea on how fast the system could be in an ideal environment. However, the precise amount of data being exchanged is reported.

Notice in Table 3.7 that the evaluator spends an unreasonable amount of time on communication with respect to the amount of data to be transmitted in both the oblivious transfer and circuit construction stages. This is because the evaluator spends that time waiting for the generator to finish computation-intensive tasks. The same reasoning explains why in the circuit evaluation stage the generator spends more time in communication than the evaluator. This waiting results from the fact that both parties need to run the protocol in a synchronized manner. A generator-evaluator pair cannot start next communication round while any other pair has not finished the current one. This synchronization is crucial since the protocol’s security is guaranteed only when each communication round is performed sequentially. While the parallelization of the program introduces high performance execution, it does not and should not change this essential property. A stronger notion of security such as universal security will be required if asynchronous communication is allowed. By using TCP sockets in “blocking” mode the implementation enforces this communication round synchronization.
Note that the low communication during the circuit construction stage is due to the random seed checking technique. Also, the fact that the generator spends more time in the evaluation stage than she traditionally does comes from the second construction for evaluation circuits. Recall that only the evaluation circuits need to be sent to the evaluator. Since only 40% of the garbled circuits (102 out of 256) are evaluation-circuits, the ratio of the generator’s computation time in the generation and evaluation stage is 35.63:14:92 ≈ 5:2.

At the time these experiments were run it was unfortunately difficult to find a cluster of hundreds of nodes that all support AES-NI. The experimental results, therefore, do not show the full potential of all the optimization techniques discussed above. However, recall that for certain circuits the running time in the semi-honest setting is roughly half of that in the malicious setting. Based on the preliminary results in Table 3.5 the performance improvement from AES-NI is approximately 20%

Table 3.8 shows that the Yao protocol really benefits from the circuit-level parallelization. Starting from Table 3.7, where each side only has one process, all the way to when each side has 256 processes, as the degree of parallelism is multiplied by four, the total time reduces into a quarter. Note that the communication costs between the generator and evaluator remain the same, as shown in Table 3.7. It may seem odd that the communication costs are reduced as the number of processes increase. The real interpretation of this data is that as the number of processes increases, the “waiting time” decreases.

Notice that as the number of processes increases, the ratio of the time the generator spends in the construction and evaluation stage decreases from 5:2 to 1:1. The reason is that the number of garbled circuit each process handles is getting smaller and smaller. Eventually, further increasing the number of processes will reach limits of the benefits that the circuit-level parallelism could possibly bring. In that case, each process is dealing with merely a single copy of the garbled circuit, and the time spent in both the generation and evaluation stages is the time to construct a garbled circuit.

Large Circuits In this experiment, the 4095-bit edit distance circuit, that is, \((x, y) \mapsto (\bot, \text{EDT}(x, y))\), where \(x, y \in \{0, 1\}^{4095}\) was tested. This experiment used the \(I + C\) approach, where the computation time could be roughly a half of that of the \(I + 2C\) approach with the price of not getting to use the random-seed technique. Recall that in the \(I + C\) approach, the generator and the evaluator conduct the cut-and-choose in a way that the generator does not know the check circuits until she finishes transferring all the garbled circuits. Next, both the parties run the circuit generation and evaluation in a pipeline manner, where one party is generating and giving away garbled gates on one end, and the other party is evaluating and checking the received gates at the other end at the same time. The results are shown in Table 3.9.

The circuit generated by the BillionYao compiler had 5.9 billion gates, and 2.4 billion of those are non-XOR. It is worth mentioning that without the random-seed technique, the communication cost shown in Table 3.9 can also be estimated by \(256 \times 2.4 \times 10^9 \times 3 \times 10 = 1.8 \times 10^{13}\), since 256 copies of the garbled circuits need to be transferred, each copy has 2.4 billion non-free gates, each non-free gate has three entries, and each entry has \(k = 80\) bits.

In addition to showing that BillionYao is capable of handling the largest circuits ever reported, these experiment achieved speeds in the malicious setting that is comparable to those in the semi-honest setting. In particular, a single execution of 4095-bit edit distance circuit required less than 8.2 hours with a rate of 82,000 (non-XOR) gates per second.
<table>
<thead>
<tr>
<th></th>
<th>Gen (sec)</th>
<th>Eval (sec)</th>
<th>Comm (Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OT</td>
<td>19.73±0.5%</td>
<td>5.26±0.4%</td>
<td>1.7 × 10^8</td>
</tr>
<tr>
<td></td>
<td>1.1±6%</td>
<td>15.6±0.6%</td>
<td></td>
</tr>
<tr>
<td>Cut-&amp;</td>
<td>1.1±0.8%</td>
<td>–</td>
<td>6.5 × 10^7</td>
</tr>
<tr>
<td>Choose</td>
<td>–</td>
<td>1.5±2%</td>
<td></td>
</tr>
<tr>
<td>Gen./Evl.</td>
<td>24,400±1%</td>
<td>14,600±3%</td>
<td>1.8 × 10^{13}</td>
</tr>
<tr>
<td></td>
<td>4,900±1%</td>
<td>14,700±2%</td>
<td></td>
</tr>
<tr>
<td>Inp.</td>
<td>0.6±20%</td>
<td>–</td>
<td>8.5 × 10^6</td>
</tr>
<tr>
<td>Chk</td>
<td>0.4±40%</td>
<td>0.60±20%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>24,400±1%</td>
<td>14,600±3%</td>
<td>1.8 × 10^{13}</td>
</tr>
<tr>
<td></td>
<td>4,900±1%</td>
<td>14,700±2%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.9: The result of \((x, y) \mapsto (\bot, \text{EDT-4095}(x, y))\). Each party is comprised of 256 cores in a cluster. This table comes from 6 invocations of the system. Similarly, the upper row in each stage is the computation time, while the lower is the communication time.

### 3.7 Chapter Acknowledgements

The BillionYao system was joint work with Chih-hao Shen and abhi shelat. We would like to thank Benny Pinkas, Thomas Schneider, Nigel Smart and Stephen Williams for providing us with a copy of their optimized AES circuit. We would also like to thank Gabriel Robins for his advice on minimizing circuits in VLSI systems. We are particularly grateful to Ian Goldberg for his very helpful comments.
Chapter 4

PCF: A Portable Circuit Format For Scalable Two-Party Secure Computation

While the system presented in Chapter 3 can handle large circuits, and can perform automatic optimizations on large circuits quite efficiently, its scalability is limited by its storage requirements. In this chapter the more scalable PCF system is described, with reduced memory requirements and a more powerful compiler. PCF does not use a domain-specific language like BillionYao, but instead cross-compiles the intermediate format of a C compiler, allowing users to specify functionality using C (with certain necessary restrictions describe in more detail below). This chapter is based on the work of Kreuter et al. [KSMB13].

4.1 Bytecode

A common approach to compiler design is to translate a high level language into a sequence of instructions for a simple, abstract machine architecture; this is known as the intermediate representation or bytecode. Bytecode representations have the advantage of being machine-independent, thus allowing a compiler front-end to be used for multiple target architectures. Optimizations performed on bytecode are machine independent as well; for example, dead code elimination is typically performed on bytecode, as removing dead code causes programs to run faster on all realistic machines.

This chapter focuses on a commonly used bytecode abstraction, the stack machine. In this model, operands must be pushed onto an abstract stack, and operations involve popping operands off of the stack and pushing the result. In addition to the stack, a stack machine has RAM, which is accessed by instructions that pop an address off the stack. Instructions in a stack machine are partially ordered, and are divided into subroutines in which there is a total ordering. In addition to simple operations and operations that interact with RAM, a stack machine has operations that can modify the program counter, a pointer to the next instruction to be executed, either conditionally or unconditionally.

At a high level, the PCF system translates bytecode programs for a stack machine into boolean circuits for SFE. At first glance, this would appear to be at least highly inefficient, if not impossible, because of the many ways such an input program could loop. In fact, imposing only a small set of restrictions on permissible sequences of instructions enables an efficient and practical translator, without significantly
Figure 4.1: The high-level concept of the PCF design. It is not necessary to unroll loops at compile time, even to perform optimizations on the circuit. Instead, loops can be evaluated at runtime, with gates being computed on-the-fly, and loop indices being updated locally by each party. Wire values are stored in a table, with each gate specifying which two table entries should be used as inputs and where the output should be written; previous wire values in the table can be overwritten during this process, if they are no longer needed.

reducing the usability or expressive power of the high level language.

4.2 System Design

The PCF system divides the compiler into several stages, following a common compiler design. For testing, the LCC compiler front end was used to parse C source code and produce a bytecode intermediate representation (IR). The PCF compiler back end performs optimizations and translates the bytecode into a description of a secure computation protocol using a custom format. This representation greatly reduces the disk space requirements for large circuits compared to previous work, while still allowing optimizations to be done at the bit level. The PCF compiler is written in Common Lisp, using the Steel Bank Common Lisp system.

4.2.1 Compact Representations of Boolean Circuits

In Fairplay and the systems that followed its design, the common pattern has been to represent Boolean circuits as adjacency lists, with each node in the graph being a gate. The introduces a scalability problem, as it requires storage proportional to the size of the circuit. Generating, optimizing, and storing circuits has been a bottleneck for previous compilers, even for relatively simple functions like RSA. Loading such large circuits into RAM is a challenge, as even very high-end machines may not have enough RAM for relatively simple functions.

There have been some approaches to addressing this scalability problem presented in previous work. The BillionYao system reduced the RAM required for protocol executions by assigning each gate’s output wire a reference count, allowing the memory used for a wire value to be deallocated once the gate is no longer needed. However, the compiler bottleneck is not solved in BillionYao, as even computing the reference count required memory proportional to the size of the circuit. Even with the engineering
improvements presented in Chapter 3, the BillionYao compiler is unable to compile circuits with more than a few billion gates, and requires several days to compile the largest test cases.

The PAL system [MLB12b] also addresses memory requirements, by adding control structures to the circuit description, allowing parts of the description to be re-used. In the original presentation of PAL, however, a large circuit file would still be emitted in the Fairplay format when the secure protocol was run. An extension of this work presented by Mood [Mooon] allowed the PAL description to be used directly at runtime, but this work sacrificed the ability to optimize circuits automatically.

The PCF system builds upon the PAL and BillionYao systems to solve the memory scalability problem without sacrificing the ability to optimize circuits automatically. Two observations are key to PCF’s design.

A first observation is that it is possible to free the memory required for storing wire values without computing a reference count for the wire. In previous work, each wire in a circuit is assigned a unique global identifier, and gate input wires are specified in terms of these identifiers (output wires can be identified by the position of the gate in the gate list). Rather than using global identifiers, we observe that wire values are ephemeral, and only require a unique identity until their last use as the input to a gate.

PCF therefore maintains a table of “active” wire values, similar to BillionYao, but change the gate description. In this format, wire values are identified by their index in the table, and gates specify the index of each input wire and an index for the output wire; in other words, a gate is a tuple \((t, i_1, i_2, o)\), where \(t\) is a truth table, \(i_1, i_2\) are the input wire indexes, and \(o\) is the output wire index. When a wire value is no longer needed, its index in the table can be safely used as an output wire for another gate.

Now, consider the following example of a circuit described in the above format, which accumulates the Boolean AND of seven wire values:

\[
\langle AND_1, 1, 2, 0 \rangle \\
\langle AND_2, 0, 3, 0 \rangle \\
\langle AND_3, 0, 4, 0 \rangle \\
\langle AND_4, 0, 5, 0 \rangle \\
\langle AND_5, 0, 6, 0 \rangle \\
\langle AND_6, 0, 7, 0 \rangle
\]

A second observation is that circuits such as this can be described more compactly using a loop structure. This builds on the first observation, which allows wire values to be overwritten once they are no longer needed. A simple approach to allowing this would add a conditional branch operation to the description format. This is more general than the approach of PAL, which includes loops but allows only simple iteration. Additionally, it is necessary to allow the loop index to be used to specify the input or output wire index of the gates; as a general solution, PCF supports indirection, allowing wire values to be copied.

PCF’s representation of Boolean circuits is a bytecode for a one-bit CPU, where the operations are the 16 possible two-arity Boolean gates, a conditional branch, and indirect copy. PCF also includes instructions for function calls (which need not be inlined at compile time) and handling the parties’ inputs/outputs. When the secure protocol is run, a three-level logic is used for wire values: 0, 1, or \(\perp\), where \(\perp\) represents an “unknown” value that depends on one of the party’s inputs. In the case of a Yao
protocol, the \( \perp \) value is represented by a garbled wire value. Conditional branches are not allowed to depend on \( \perp \) values, and indirection operations use a separate table of pointers that cannot computed from \( \perp \) values (if such an indirection operation is required, it must be translated into a large multiplexer, as in previous work).

### 4.3 PCF Semantics

The PCF file format consists of a header section that declares the input size, followed by a list of operations that are divided into subroutines. At runtime, these operations manipulate the internal state of the PCF interpreter, causing gates to be emitted when necessary. The internal state of the PCF interpreter consists of an instruction pointer, a call stack, an array of wire values, and an array of pointers. The pointers are positive integers. Wire values are 0, 1, or \( \perp \), where \( \perp \) represents a value that depends on input data, which is supplied by the code that invokes the interpreter. Each position in the wire table can be treated as a stack.

Each PCF instruction can take up to 3 arguments. The instructions and their semantics are as follows:

**CLABEL/SETLABEL** Appears only in the header, used for setting the input size for each party. 
  CLABEL declares the bit width of a value, SETLABELC sets the value.

**FUNCTION** Denotes the beginning of a subroutine. When the subroutine is called, the instruction pointer is set to the position following this instruction.

**GADGET** Denotes a branch target

**BRANCH** Takes two arguments: a target, declared with GADGET, and a location in the wire table. In the wire value is 0, the instruction pointer is set to the instruction following the target. If the wire value is 1, the instruction pointer is incremented. If the wire value is \( \perp \), evaluation halts with an error.

**FUNC** Calls a subroutine, pushing the current instruction pointer onto the call stack.

**PUSH** Pushes a copy of the wire value at a specified position onto the stack at that position.

**POP** Pops a stack at a specified position. If there is only one value on that stack, evaluation halts with an error.

**ALICEIN32/BOBIN32** Fetches 32 input bits from one party, beginning at a specified bit position in that party’s input. The bit position is specified by an array of 32 values in the wire table. If any of the values is \( \perp \), evaluation halts with an error. The input values will all have the value \( \perp \), and will be stored in the wire table at positions 0 through 31.

**SHIFT OUT** Outputs a single bit for a given party

**RETURN** Return from a subroutine. The instruction pointer is repositioned to the value popped from the top of the call stack.

**STORECONSTPTR** Sets a value in the pointer table
OFFSETPTR  Adds a value to a pointer, specified by an array of 32 wire values starting at a position in the wire table. If any value in the array is ⊥, evaluation halts with an error.

PTRTOWIRE  Saves a pointer value as a 32 bit unsigned integer. Each of the bits is pushed onto the stack at a location in the wire table.

PTRTOPTR  Copies a value from one position in the pointer table to another.

CPY121 Copy a wire value from a position specified by a pointer to a statically specified position.

CPY32 Copy a wire value from a statically specified position to a position specified by a pointer.

Compute a gate with the specified truth table on two input values from the wire table, with output stored at a specified position. Logic simplification rules are applied when one or both of the input values is ⊥. If no simplification is possible, then the output will be ⊥ and the interpreter will emit a gate. This is used for both local computations such as updating a loop index, and for computing the gates used by the protocol.

4.3.1 Example PCF Description

Below is an example of a PCF file. It iterates over a loop several times, XORing the two parties’ inputs with a bit from the internal state.

GADGET: main
CLABEL ALICEINLENGTH 32
CLABEL BOBINLENGTH 32
CLABEL xxx 32
SETLABELC ALICEINLENGTH 128
SETLABELC ALICEINLENGTH 128
FUNCTION: main
1111 32 0 0
0000 33 0 0
0000 34 0 0
0000 35 0 0
GADGET: L
0110 36 35 34
0001 35 36 36
0110 36 34 33
0001 34 36 36
0110 36 33 32
0001 33 36 36
ALICEINPUT32 0 0
0001 36 0 0
BOBININPUT32 0 0
0001 37 0 0
0110 38 37 36
4.3.2 Describing Functions for SFE

Most commonly used programming languages can describe processes that cannot be translated to SFE; for example, a program that does not terminate, or one which terminates after reading a specific input pattern. It is therefore necessary to impose some limitation on the descriptions of functions for SFE. In systems with domain specific languages, these limitations can be imposed by the grammar of the language, or can be enforced by taking advantage of particular features of the grammar. However, one goal of the PCF system is to allow any programming language to be used to describe functionality for SFE, and so it cannot rely on the grammar of the language being used.

PCF makes a compromise when it comes to restricting the input source code. Unlike model checking systems [BCCZ99], it imposes no upper bound on loop iterations or on recursive function calls (other than the memory available for the call stack), and leaves the responsibility of ensuring that programs terminate to the programmer. On the other hand, PCF does forbid certain easily-detectable conditions that could result in infinite loops, such as unconditional backwards jumps, conditional backwards jumps that depend on input, and indirect function calls. These restrictions are similar to those imposed by the Fairplay and BillionYao systems [MNPS04, KSS12], but allow for more general iteration than incrementing the loop index by a constant. Although false positives i.e. programs that terminate but which contain such constructs are possible, the design hypothesis is that useful functions and typical compilers would not result in such instruction sequences, and in experiments using LCC no such sequences were observed.

4.3.3 Algorithms for Translating Bytecode

The PCF compiler reads a bytecode representation of the function, which lacks the structure of higher-level descriptions and poses a unique challenge in circuit generation. As mentioned above, it does not impose any upper limit on loop iterations or the depth of the function call stack. The PCF compiler’s approach to translation does not use any symbolic analysis of the function. Instead, it translates the bytecode into the PCF format, using conditional branches and function calls as needed and translating other instructions into lists of gates. The experimental implementation reads the IR from the LCC compiler, which is based on the common stack machine model; examples of this IR are used in the following sections to illustrate this design, but note that none of these techniques strictly require a stack machine model or any particular features of the LCC bytecode.

The PCF compiler divides bytecode instructions into three classes:

**Normal** Instructions which have exactly one successor and which can be represented by a simple circuit. Examples of such instructions are arithmetic and bitwise logic operations, operations that push data onto the stack or move data to memory, etc.

**Jump** Instructions that result in an unconditional control flow switch to a specific label. This does not include function calls, which are represented directly in PCF. Such instructions are usually used for if/else constructs or preceding the entry to a loop.
Conditional Instructions that result in control flow switching to either a label or the subsequent instruction, depending on the result of some conditional statement. Examples include arithmetic comparisons.

In the stack machine model, all operands and the results of operations are pushed onto a global stack. For “normal” instructions, the translation procedure is straightforward: the operands are popped off the stack and assigned temporary wires, the subcircuit for the operation is connected to these wires, and the output of the operation is pushed onto the stack. “Jump” instructions appear, at first, to be equally straightforward, but actually require special care as we describe below.

“Conditional” instructions present a challenge. Conditional jumps whose targets precede the jump are assumed to be loop constructs, and are translated directly into PCF branch instructions. All other conditional jumps require the creation of multiplexers in the circuit to deal with conditional assignments. Therefore, the branch targets must be tracked to ensure that the appropriate condition wires are used to control those multiplexers.

In the Fairplay and BillionYao compilers, the condition wire for an “if” statement is pushed onto a stack along with a “scope” that is used to track the values (wire assignments) of variables. When a conditional block is closed, the condition wire at the top of the stack is used to multiplex the value of all the variables in the scope at the top with the values from the scope second to the top, and then the stack is popped. This procedure relies on the grammar of “if/else” constructs, which ensures that conditional blocks can be arranged as a tree. An example of this type of “if/else” construct is in Figure 4.2. In a bytecode representation, however, it is possible for conditional blocks to “overlap” with each other without being nested.

In the sequence shown in Figure 4.3, the first branch’s target precedes the second branch’s target, and indirect loads and assignments exist in the overlapping region of these two branches. The control flow of such an overlap is given in Figure 4.4. A stack is no longer sufficient in this case, as the top of the stack will not correspond to the appropriate branch when the next branch target is encountered. Such instruction sequences are not uncommon in the code generated by production compilers, as they are a convenient way to generate code for “else” blocks and ternary operators.

To handle such sequences, PCF uses an algorithm based on a priority queue rather than a stack, and maintains a global condition wire that is modified as branches and branch targets are reached. When a branch instruction is reached, the global condition wire is updated by logically ANDing the branch condition with the global condition wire. The priority queue is updated with the branch condition and a scope, as in the stack-based algorithm; the priority is the target, with lower targets having higher priority. When an assignment is performed, the scope at the top of the priority queue is updated with the value being assigned, the location being assigned to, the old value, and a copy of the global condition wire. When a branch target is reached, multiplexers are emitted for each assignment recorded in the scope at the top of the priority queue, using the copy of the global condition wire that was recorded. After the
Figure 4.3: A bytecode sequence where overlapping conditional blocks are not nested; note that the target of the first branch, “A,” precedes the target of the second branch, “B.”

Figure 4.4: A control flow with overlapping conditional blocks.

multiplexers are emitted, the global condition wire is updated by ORing the inverse of the condition wire at the top of the priority queue, and then the top is removed.

Unconditional jumps are only allowed in the forward direction, i.e., only if the jump precedes its target. When such instructions are encountered, they are translated into conditional branches whose condition wire is the inverse of the conjunction of the condition wires of all enclosing branches. In the case of a jump that is not in any conditional block, the condition wire is set to false; this does not necessarily mean that subsequent assignments will not occur, as the multiplexers for these assignments will be emitted and will depend on a global control line that may be updated as part of a loop construct. The optimizer is responsible for determining whether such assignments can occur, and will rewrite the multiplexers as direct assignments when possible.

Finally, it is possible that the operand stack will have changed in the fall-through path of a conditional jump. In that case, the stack itself must be multiplexed. For simplicity, PCF requires that the depth of the stack not change in a fall-through path. This requirement was not violated in the IR generated by LCC.

4.3.4 Optimization

One of the shortcomings of the BillionYao system is the amount of time and memory required to perform optimizations on the computed circuit. In the PCF system, optimization is performed before loops are unrolled but after the functionality is translated into a PCF representation. This allows optimizations to be performed on a smaller representation, but increases the complexity of the optimization process somewhat.

The BillionYao compiler bases its optimization on a rudimentary dataflow analysis, but without any conditional branches or loops, and with single assignments to each wire. In PCF, loops are not eliminated and wires may be overwritten, but conditional branches are eliminated. As in BillionYao an approach based on dataflow analysis is employed, but the PCF compiler make multiple passes to find a fixed point solution to the dataflow equations. The dataflow equations take advantage of the logical rules of each gate, allowing more gates to be identified for elimination than the textbook equations identify.

The dataflow analysis is performed on individual PCF instructions, which allows us to remove single
gates even where entire bytecode instructions could not be removed, but which carries the cost of somewhat longer compilation time, on the order of minutes for the experiments presented below. The experimental implementation only performs optimization within individual functions, without any interprocedural analysis. Compile times in this system can be reduced by splitting a large procedure into several smaller procedures.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>128 mult.</th>
<th>5x5 matrix</th>
<th>256 RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>707,244</td>
<td>260,000</td>
<td>904,171,008</td>
</tr>
<tr>
<td>Const. Prop.</td>
<td>296,960</td>
<td>198,000</td>
<td>651,504,495</td>
</tr>
<tr>
<td>Dead Elim.</td>
<td>700,096</td>
<td>255,875</td>
<td>883,307,712</td>
</tr>
<tr>
<td>Both</td>
<td>260,073</td>
<td>131,875</td>
<td>573,156,735</td>
</tr>
</tbody>
</table>

Table 4.1: Effects of constant propagation and dead code elimination on circuit size, measured with simulator that performs no simplification rules. For each function, the number of non-XOR gates are given for all combinations of optimizations enabled.

**Constant Propagation**

The constant propagation framework is straightforward, similar to the methods used in typical compilers. However, for some gates, simplification rules can result in constants being computed even when the inputs to a gate are not constant; for example, XORing a variable with itself. The transfer function is augmented with a check against logic simplification rules to account for this situation, but remains monotonic and so convergence is still guaranteed.

**Dead Gate Removal**

The last step of the optimizer is to remove gates whose output wires are never used. This is a standard bit vector dataflow problem that requires little tailoring. As is common in compilers, performing this step last yields the best results, as large numbers of gates become dead following earlier optimizations.

**4.3.5 Externally-Defined Functions**

Some functionality is difficult to describe well in bytecode formats. For example, the graph isomorphism experiment presented in Section 4.5 uses AES as a PRNG building block, but the best known description of the AES S-box is given at the bit-level [BP10], whereas the smallest width operation supported by LCC is a single byte. To compensate for this difficulty, PCF allows users to specify functions with the same language used internally to translate bytecode operations into circuits; an example of this language is shown in Section 4.4.1. This allows for possible combinations of the compiler with other circuit generation and optimization tools.

**4.3.6 PCF Interpreter**

To use a PCF description of a circuit in a secure protocol, an interpreter is needed. The interpreter simulates the execution of the PCF file for a single-bit machine, emitting gates as needed for the protocol. Loops are not explicitly unrolled; instead, PCF branch instructions are conditionally followed, based on the logic value of some wire, and each wire identifier is treated as an address in memory. This is where the
<table>
<thead>
<tr>
<th>Function</th>
<th>With</th>
<th>Without</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384-bit Comp.</td>
<td>32,228</td>
<td>49,314</td>
<td>65%</td>
</tr>
<tr>
<td>128-bit Sum</td>
<td>345</td>
<td>508</td>
<td>67%</td>
</tr>
<tr>
<td>256-bit Sum</td>
<td>721</td>
<td>1,016</td>
<td>70%</td>
</tr>
<tr>
<td>1024-bit Sum</td>
<td>2,977</td>
<td>4,064</td>
<td>73%</td>
</tr>
<tr>
<td>128-bit Mult.</td>
<td>76,574</td>
<td>260,073</td>
<td>20%</td>
</tr>
<tr>
<td>256-bit Mult.</td>
<td>300,634</td>
<td>1,032,416</td>
<td>20%</td>
</tr>
<tr>
<td>1024-bit Mult.</td>
<td>8,301,962</td>
<td>19,209,120</td>
<td>21%</td>
</tr>
</tbody>
</table>

Table 4.2: Non-XOR gates in circuits computed by the interpreter with and without the application of simplification rules by the runtime system.

requirement that loop bounds be independent of both parties’ inputs is ultimately enforced: the interpreter cannot determine whether or not to take a branch if it cannot determine the condition wire’s value.

### 4.3.7 Threat Model

The PCF system treats the underlying secure computation protocol as a black box, without making any assumptions about the threat model. In Section 4.5, running times for smaller circuits in the malicious model version of the BillionYao protocol are presented. This malicious model implementation simply invokes multiple copies of the same PCF interpreter used for the semihonest version, one for each copy of the circuit needed in the protocol.

### 4.3.8 Runtime Optimization

Some optimizations cannot be performed without unrolling loops, and so PCF defers these optimizations until the program is interpreted. As an example, logic simplification rules that eliminate gates whose output values depend on no more than one of their input wires can only be partially applied at compile time, as some potential applications of these rules might only be possible for some iterations of a loop. While it is possible to compute this information at compile time, in the general case this would involve storing information about each gate for every iteration of every loop, which would be as expensive as unrolling all loops at compile time.

A side effect of applying such logic simplification rules is automatic copy propagation. A gate that always takes on the same value as one of its inputs is equivalent to a copy operation. The application of logic simplification rules to such a gate results in the interpreter simply copying the value of the input wire to the output wire, without emitting any gate. As there is little overhead resulting from the application of simplification rules at runtime, compile times are further reduced by delaying this optimization until runtime.

For each gate, the interpreter checks if the gate’s value can be statically determined, i.e., if its output value does not rely on either party’s input bits. This is critical, as some of the gates in a PCF file are used for control flow, e.g., to increment a loop index. Additionally, logic simplification rules are applied where possible in the interpreter. This allows the interpreter to not emit gates that follow an input or which have static outputs even when their inputs cannot be statically determined. As shown in Table 4.2, in some cases up to 80% of the gates could be removed in this manner. Even in a simulator that performs no garbling, applying this runtime optimization not only shows no performance overhead, but actually a very slight performance gain, as shown in Table 4.3. The slight performance gain is a result of the transfer of
### Table 4.3: Simulator time with simplification rules versus without, using the C interpreter. Times are averaged over 50 samples, with 95% confidence intervals, measured using the `time` function implemented by SBCL.

<table>
<thead>
<tr>
<th>Function</th>
<th>With (s)</th>
<th>Without (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384-bit Comp.</td>
<td>4.41 ± 0.3%</td>
<td>4.44 ± 0.3%</td>
</tr>
<tr>
<td>128-bit Sum</td>
<td>0.0581 ± 0.3%</td>
<td>0.060 ± 2%</td>
</tr>
<tr>
<td>256-bit Sum</td>
<td>0.103 ± 0.3%</td>
<td>0.105 ± 0.3%</td>
</tr>
<tr>
<td>1024-bit Sum</td>
<td>0.365 ± 0.3%</td>
<td>0.367 ± 0.2%</td>
</tr>
<tr>
<td>128-bit Mult.</td>
<td>0.892 ± 0.1%</td>
<td>0.894 ± 0.1%</td>
</tr>
<tr>
<td>256-bit Mult.</td>
<td>3.02 ± 0.1%</td>
<td>3.04 ± 0.1%</td>
</tr>
<tr>
<td>1024-bit Mult.</td>
<td>39.7 ± 0.2%</td>
<td>39.9 ± 0.06%</td>
</tr>
</tbody>
</table>

### Table 4.4: Comparisons between the PCF compiler’s output and the output of the BillionYao and Holzer et al. (HFKV) compilers, in terms of non-XOR gates.

<table>
<thead>
<tr>
<th>Function</th>
<th>This Work</th>
<th>KSS12</th>
<th>HFKV</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384 Comp.</td>
<td>32,229</td>
<td>49,149</td>
<td>-</td>
</tr>
<tr>
<td>RSA 256</td>
<td>235,925,023</td>
<td>332,085,981</td>
<td>-</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>880</td>
<td>-</td>
<td>3,003</td>
</tr>
<tr>
<td>Hamming 1600</td>
<td>9,625</td>
<td>-</td>
<td>30,318</td>
</tr>
<tr>
<td>3x3 Matrix</td>
<td>27,369</td>
<td>160,949</td>
<td>47,871</td>
</tr>
<tr>
<td>5x5 Matrix</td>
<td>127,225</td>
<td>746,177</td>
<td>221,625</td>
</tr>
<tr>
<td>8x8 Matrix</td>
<td>522,304</td>
<td>3,058,754</td>
<td>907,776</td>
</tr>
<tr>
<td>16x16 Matrix</td>
<td>4,186,368</td>
<td>24,502,530</td>
<td>7,262,208</td>
</tr>
</tbody>
</table>

control that occurs when a gate is emitted, which has a small but non-trivial cost in the simulator. In a garbled circuit protocol, this cost would be even higher, because of the time spent garbling gates.

## 4.4 Portability

### 4.4.1 Portability Between Bytecodes

The PCF compiler can be given a description of how to translate bytecode instructions into boolean circuits using a special internal language. An example, for the LCC instruction “`ADDU`,” is shown in Figure 4.5. The first line is specific to LCC, and would need to be modified for use with other front-ends. The second line assumes a stack machine model: this instruction reads two instructions from the stack. Following that is the body of the translation rule, which can be used in general to describe circuit components and how the input variables should be connected to those components.

The description follows an abstraction similar to VMCrypt, in which a unit gadget is “chained” to create a larger gadget. It is possible to create chains of chains, e.g., for a shift-and-add multiplier as well. For more complex operations, Lisp source code can be embedded, which can interact directly with the compiler’s internal data structures.

### 4.4.2 Portability Between SFE Systems

Both the PCF compiler and the interpreter can treat the underlying secure computation system as a black box. Switching between secure computation systems, therefore, requires work only at the “back end” of the interpreter, where gates are emitted. Two possible approaches were implemented for the tests:
### Table 4.5: Summary of circuit sizes for various functions and the time required to compile and interpret the PCF files in a protocol simulator. Times are averaged over 50 samples, with 95% confidence intervals, except for RSA-1024 simulator time, which is averaged over 8 samples. Run times were measured using the time function implemented in SBCL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Total Gates</th>
<th>non-XOR Gates</th>
<th>Compile Time (s)</th>
<th>Simulator Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384-bit Comp.</td>
<td>97,733</td>
<td>32,229</td>
<td>3.40 ± 4%</td>
<td>4.40 ± 0.2%</td>
</tr>
<tr>
<td>Hamming 160</td>
<td>4,368</td>
<td>880</td>
<td>9.81 ± 1%</td>
<td>0.0810 ± 0.3%</td>
</tr>
<tr>
<td>Hamming 1600</td>
<td>32,912</td>
<td>6,375</td>
<td>11.0 ± 0.4%</td>
<td>0.52 ± 8%</td>
</tr>
<tr>
<td>Hamming 16000</td>
<td>389,312</td>
<td>97,175</td>
<td>10.8 ± 0.2%</td>
<td>4.83 ± 0.5%</td>
</tr>
<tr>
<td>128-bit Sum</td>
<td>1,443</td>
<td>345</td>
<td>4.70 ± 3%</td>
<td>0.0433 ± 0.4%</td>
</tr>
<tr>
<td>256-bit Sum</td>
<td>2,951</td>
<td>721</td>
<td>4.60 ± 3%</td>
<td>0.0732 ± 0.4%</td>
</tr>
<tr>
<td>1024-bit Sum</td>
<td>11,999</td>
<td>2,977</td>
<td>4.60 ± 3%</td>
<td>0.250 ± 0.5%</td>
</tr>
<tr>
<td>64-bit Mult.</td>
<td>105,880</td>
<td>24,766</td>
<td>71.7 ± 0.2%</td>
<td>0.332 ± 0.4%</td>
</tr>
<tr>
<td>128-bit Mult.</td>
<td>423,064</td>
<td>100,250</td>
<td>74.9 ± 0.1%</td>
<td>0.903 ± 0.3%</td>
</tr>
<tr>
<td>256-bit Mult.</td>
<td>1,659,808</td>
<td>400,210</td>
<td>79.5 ± 0.9%</td>
<td>3.07 ± 0.2%</td>
</tr>
<tr>
<td>1024-bit Mult.</td>
<td>25,592,368</td>
<td>6,371,746</td>
<td>74.0 ± 0.2%</td>
<td>40.9 ± 0.4%</td>
</tr>
<tr>
<td>256-bit RSA</td>
<td>673,105,990</td>
<td>235,925,023</td>
<td>381. ± 0.2%</td>
<td>98.0 ± 0.3%</td>
</tr>
<tr>
<td>512-bit RSA</td>
<td>5,397,821,470</td>
<td>1,916,813,808</td>
<td>350. ± 0.2%</td>
<td>7,330 ± 0.2%</td>
</tr>
<tr>
<td>1024-bit RSA</td>
<td>42,151,698,718</td>
<td>15,149,856,895</td>
<td>564. ± 0.2%</td>
<td>56,000 ± 0.3%</td>
</tr>
<tr>
<td>3x3 Matrix Mult.</td>
<td>92,961</td>
<td>27,369</td>
<td>306. ± 1%</td>
<td>0.256 ± 0.5%</td>
</tr>
<tr>
<td>5x5 Matrix Mult.</td>
<td>433,475</td>
<td>127,225</td>
<td>343. ± 0.7%</td>
<td>0.94 ± 2%</td>
</tr>
<tr>
<td>8x8 Matrix Mult.</td>
<td>1,782,656</td>
<td>522,304</td>
<td>109. ± 0.1%</td>
<td>3.14 ± 0.3%</td>
</tr>
<tr>
<td>16x16 Matrix Mult.</td>
<td>14,308,864</td>
<td>4,186,368</td>
<td>109. ± 0.1%</td>
<td>23.7 ± 0.3%</td>
</tr>
</tbody>
</table>

Table 4.6: Times of HFKV and BillionYao compilers with circuit sizes. The Mult. program uses a Shift-Add implementation. All times are averaged over 50 samples with the exception of the HFKV 256-bit multiplication, which was run for 10 samples; times are given with 95% confidence intervals.

<table>
<thead>
<tr>
<th>Function</th>
<th>Total Gates</th>
<th>non-XOR gates</th>
<th>Time (s)</th>
<th>Total Gates</th>
<th>non-XOR gates</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384-bit Comp.</td>
<td>330,784</td>
<td>131,103</td>
<td>10.5 ± 0.1%</td>
<td>98,303</td>
<td>49,154</td>
<td>4.66 ± 0.5%</td>
</tr>
<tr>
<td>3x3 Matrix Mult.</td>
<td>172,315</td>
<td>47,871</td>
<td>2.2 ± 4%</td>
<td>424,748</td>
<td>160,949</td>
<td>10.5 ± 0.5%</td>
</tr>
<tr>
<td>5x5 Matrix Mult.</td>
<td>797,751</td>
<td>221,625</td>
<td>8.40 ± 0.3%</td>
<td>1,968,452</td>
<td>746,177</td>
<td>48.2 ± 0.2%</td>
</tr>
<tr>
<td>8x8 Matrix Mult.</td>
<td>3,267,585</td>
<td>907,776</td>
<td>59.4 ± 0.3%</td>
<td>8,067,458</td>
<td>3,058,754</td>
<td>210 ± 2%</td>
</tr>
<tr>
<td>16x16 Matrix Mult.</td>
<td>26,140,673</td>
<td>7,262,208</td>
<td>2,600 ± 7%</td>
<td>64,570,969</td>
<td>24,502,530</td>
<td>2,200 ± 1%</td>
</tr>
<tr>
<td>32-bit Mult.</td>
<td>65,121</td>
<td>26,624</td>
<td>6.43 ± 0.3%</td>
<td>15,935</td>
<td>5,983</td>
<td>0.33 ± 3%</td>
</tr>
<tr>
<td>64-bit Mult.</td>
<td>321,665</td>
<td>126,529</td>
<td>71.4 ± 0.3%</td>
<td>64,639</td>
<td>24,384</td>
<td>1.6 ± 2%</td>
</tr>
<tr>
<td>128-bit Mult.</td>
<td>1,409,025</td>
<td>451,182</td>
<td>999 ± 0.1%</td>
<td>260,351</td>
<td>97,663</td>
<td>6.10 ± 0.6%</td>
</tr>
<tr>
<td>256-bit Mult.</td>
<td>5,880,833</td>
<td>2,264,860</td>
<td>16,000 ± 2%</td>
<td>1,044,991</td>
<td>391,935</td>
<td>24.5 ± 0.2%</td>
</tr>
<tr>
<td>512-bit Mult.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4,187,135</td>
<td>1,570,303</td>
<td>105 ± 0.2%</td>
</tr>
<tr>
<td>1024-bit Mult.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16,763,518</td>
<td>6,286,335</td>
<td>430 ± 0.3%</td>
</tr>
</tbody>
</table>

1. A single function should be called when a gate should be used in the secure computation protocol. The Java implementation of PCF uses this approach, with the HEKM system.

2. Gates should be generated as if they are being read from a file, with the secure computation system calling a function. The secure computation system may need to provide “callback” functions to the PCF interpreter for copying protocol-specific data between wires. The C implementation uses this abstraction for the BillionYao system.
```
('ADDU' nil second normal nil nil
(two-stack-arg (x y) (var var)
(chain [o1 = i1 + i2 + i3,
          o2 = i1 + (i1 + i2) * (i1 + i3)]
          o2 -> i3
          x -> i1
          y -> i2
          o1 -> stack)
          (0 -> i3)))
```

Figure 4.5: Code used in the PCF compiler to map the bytecode instruction for unsigned integer addition to the subcircuit for that operation.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU (s)</th>
<th>Network (s)</th>
<th>CPU (s)</th>
<th>Network (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generator</td>
<td></td>
<td>Evaluator</td>
<td></td>
</tr>
<tr>
<td>16384-bit Comp.</td>
<td>99.8 ± 0.2%</td>
<td>5.63 ± 0.6%</td>
<td>26.0 ± 0.6%</td>
<td>79.4 ± 0.2%</td>
</tr>
<tr>
<td>Hamming 1600</td>
<td>9.13 ± 0.4%</td>
<td>0.64 ± 4%</td>
<td>2.9 ± 4%</td>
<td>6.87 ± 2%</td>
</tr>
<tr>
<td>Hamming 16000</td>
<td>91.2 ± 0.2%</td>
<td>5.67 ± 0.7%</td>
<td>28.3 ± 3%</td>
<td>69. ± 2%</td>
</tr>
<tr>
<td>64-bit Mult.</td>
<td>0.749 ± 0.3%</td>
<td>0.158 ± 0.7%</td>
<td>0.409 ± 0.3%</td>
<td>0.494 ± 0.6%</td>
</tr>
<tr>
<td>128-bit Mult.</td>
<td>2.04 ± 0.3%</td>
<td>0.52 ± 1%</td>
<td>1.25 ± 0.2%</td>
<td>1.31 ± 0.6%</td>
</tr>
<tr>
<td>256-bit Mult.</td>
<td>5.74 ± 0.5%</td>
<td>1.2 ± 2%</td>
<td>4.2 ± 2%</td>
<td>2.7 ± 3%</td>
</tr>
<tr>
<td>1024-bit Mult.</td>
<td>72.7 ± 0.2%</td>
<td>28. ± 4%</td>
<td>60. ± 2%</td>
<td>40. ± 3%</td>
</tr>
<tr>
<td>256-bit RSA</td>
<td>1940 ± 0.2%</td>
<td>767. ± 0.7%</td>
<td>1620 ± 2%</td>
<td>1080 ± 3%</td>
</tr>
<tr>
<td>1024-bit RSA</td>
<td>1.15 × 10^5 ± 0.5%</td>
<td>4.4 × 10^4 ± 4%</td>
<td>9.5 × 10^4 ± 5%</td>
<td>6.5 × 10^4 ± 7%</td>
</tr>
<tr>
<td>3x3 Matrix Mult.</td>
<td>5.33 ± 0.4%</td>
<td>0.403 ± 0.6%</td>
<td>1.45 ± 0.8%</td>
<td>4.28 ± 0.6%</td>
</tr>
<tr>
<td>5x5 Matrix Mult.</td>
<td>24.4 ± 0.2%</td>
<td>1.81 ± 0.4%</td>
<td>6.75 ± 0.9%</td>
<td>19.5 ± 0.4%</td>
</tr>
<tr>
<td>8x8 Matrix Mult.</td>
<td>100. ± 0.2%</td>
<td>7.39 ± 0.4%</td>
<td>26.8 ± 0.7%</td>
<td>81.1 ± 0.3%</td>
</tr>
</tbody>
</table>

Table 4.7: Total running time, including PCF operations and protocol operations such as oblivious transfer, for online protocols using the PCF interpreter and the BillionYao two party computation system, on two computers communicating over the University of Virginia LAN. With the exception of RSA-1024, all times are averaged over 50 samples; RSA-1024 is averaged over 8 samples. Running time is divided into time spent on computation and time spent on network operations (including blocking).

4.5 Evaluation

Several functions were used as benchmarks to test the PCF compiler, optimizer, and interpreter. The BillionYao system was used for the performance tests, with the parties communicating over a LAN. The BillionYao timings presented below are averages for the runtime for 50 runs, alternating which computer acted as the generator and which as the evaluator to account for slight configuration differences between the systems. Compiler timings are based on 50 runs of the compiler on a desktop PC with an Intel Xeon 5560 processor, 8GB of RAM, a 7200 RPM hard disk, Scientific Linux 6.3 (kernel version 2.6.32, SBCL version 1.0.38).

4.5.1 Effect of Array Sizes on Timing

Some changes in compile time can be observed as some of the functions grow larger. The dataflow analysis deals with certain pointer operations by traversing the entire local variable space of the function.
and all global memory, which in functions with large local arrays or programs with large global arrays is costly as it increases the number of wires that optimizer must analyze.

4.5.2 Experiments

Several of the functions presented in this section were tested in other systems; a comparison of the number of non-XOR gates generated by the PCF compiler and other work is presented in Table 4.4. The sizes, compile times, and interpreter times required for these circuits are listed in Table 4.5. By comparison, the compile times and circuit sizes using the BillionYao and HFKV compilers is presented in Table 4.6. As expected, the PCF compiler outperforms these previous compilers as the size of the circuits grow, due to the improved scalability of the system.

**Arbitrary-Width Millionaire's Problem** An arbitrary-width function for the millionaire’s problem serves as a simple sanity check for the PCF system. This can be viewed as a string comparison function on 32 bit characters. It outputs a 1 to the party which has the larger input. For this simple function, PCF’s performance was only slightly better than the performance of the BillionYao compiler on the same circuit.

**Matrix Multiplication** To compare PCF with the work of Holzer et al. [HFKV12], some of their experiments were duplicated, beginning with matrix multiplication on 32-bit integers. The PCF system performed favorably, particularly due to the optimizations the PCF compiler and interpreter perform. On average, PCF generated circuits that are 60% smaller. The test matrices were of size 3x3, 5x5, 8x8, and 16x16, with 32 bit integer elements.

**Hamming Distance**

Next is the Hamming distance experiment, also duplicated from Holzer et al. Again, PCF generated substantially smaller circuits. This test used input sizes of 160, 1600, and 16000 bits.

**Integer Sum** The next test is a basic arbitrary-width integer addition function, using ripple-carry addition. No array references are needed, and so the compiler easily handles this function even for very large input sizes. Input sizes of 128, 256, and 1024 bits were tested.

**Integer Multiplication** Building on the integer addition function, the next test is an integer multiplication function that uses the textbook shift-and-add algorithm. Unlike the integer sum and hamming distance functions, the multiplication function requires arrays for both input and output, which slows the compiler down as the problem size grows. Input sizes of 64, 128, 256, and 1024 were tested.

**RSA (Modular Exponentiation)** In the BillionYao system [KSS12], it was possible to compile an RSA circuit for toy problem sizes, and it took over 24 hours to compile a circuit for 256-bit RSA. This lengthy compile time and large memory requirement stems from the fact that all loops are unrolled before any optimization is performed, resulting in a very large intermediate representation to be analyzed. As a demonstration of the improvement PCF approach represents, not only were toy RSA sizes tested, but also an RSA-1024 circuit, using only modest computational resources. This experiment tested sizes of 256, 512, and 1024.

4.5.3 Online Running Times

A modified version the BillionYao implementation using the PCF interpreter was used to benchmark the online performance of PCF. These experiments used two computers running ScientificLinux 6.3, a four core Intel Xeon E5506 2.13GHz CPU, and 8GB of RAM. No time limit on computation was imposed on these machines, which was necessary to run the RSA-1024 function, which requires a little less than two
Table 4.8: Online running time in the malicious model for several circuits. Times are averaged over 50 samples, with 95% confidence intervals.

days. To compensate for slight configuration differences between the two systems, the machines used for each party were alternated between each trial.

The results of these experiments appear in Table 4.7. Note that while the simulator times given in Table 4.5 are more than half the CPU time measured, they are also on par with the time spent waiting on the network. Non-blocking I/O or a background thread for the PCF interpreter may improve performance somewhat, which is an ongoing engineering task in the implementation.

4.5.4 Malicious Model Tests

The PCF system is not limited to the semi-honest model. Preliminary results in the malicious model version of BillionYao appear below in Table 4.8. These experiments were run on the same test systems as above, using two cores for each party. The increased running times are expected, as the trials used only two cores per party. In the case of 16384-bit comparison, the increase is very dramatic, due to the large amount of time spent on oblivious transfer (as both parties have long inputs).

4.6 Chapter Acknowledgments

This chapter presents join work with Benjamin Mood, Kevin Butler, and abhi shelat. We would like to thank Elaine Shi for her helpful advice. We also thank Chih-hao Shen for his help with porting BillionYao to use PCF. Benjamin Terner provided several patches, bugfixes, and performance improvements to the PCF2 open source project.
Chapter 5

Geppetto: Versatile Verifiable Computation

In Chapter 4 scalability was achieved by viewing garbled circuits not as “circuits” but instead as programs running in a virtual machine where certain opcodes correspond to the garbled gates. This chapter presents a system called Geppetto which is inspired by a similar approach, in the setting of verifiable computation. Unlike garbled circuits, in the setting of verifiable computation there is no need to hide intermediate states. Geppetto takes advantage of this fact. This chapter is based on the work of Costello et al. [CFH+15].

5.1 Overview

Geppetto builds on the techniques used in the earlier Pinocchio system [PGHR13], which was based on an algebraic representation of circuits called Quadratic Arithmetic Programs (QAPs). Unfortunately, as with an explicitly specified circuit, QAPs can become quite large as loops are unrolled, limiting Pinocchio’s scalability. Unlike the setting of MPC, where an oblivious representation of the computation is required, in the setting of SNARKs there is no need to hide the control-flow information. Geppetto takes advantage of this fact to address Penocchio’s scalability problems, by proving that each basic block is evaluated correctly, and bootstrapping these proofs by having each basic block check the previous block’s proof.¹

5.1.1 MultiQAPs

MultiQAP Intuition

At a high level, prior verifiable computation systems like Pinocchio [PGHR13] allow a prover to convince a skeptical verifier that \( F(\vec{u}) = \vec{y} \), where \( \vec{u} \) is a verifier-supplied vector of inputs. The prover accomplishes this with a constant-sized proof \( \pi \), and the verifier’s work scales linearly in \( |\vec{u}| + |\vec{y}| \), regardless of the complexity of \( F \). However, as \( F \) grows to encompass larger and more complex functionality (see Figure 5.1), the CPU and memory costs for the prover (as well as its key size) increase superlinearly. As §5.6.2 shows, this limits prior systems to modest application parameters.

¹In the full version of Geppetto, other techniques are used to avoid the cost of bootstrapping; these are beyond the scope of this thesis, but details can be found in the full Geppetto paper [CFH+15].
Figure 5.1: MultiQAPs (a) Most previous verifiable computation systems compiled programs to a single large circuit-like representation, leading to internal redundancy. (b) By extracting common substructures, one can represent a program as an assembly of smaller circuits, but the verifier must now also check all connections between circuits. (c) MultiQAPs connect circuits using bus structures that support succinct and efficient commitments to the bus values.

To scale to larger problems, a natural approach is to decompose the proof of $F$ into a conjunction of proofs of $m$ simpler functions $F_0, \ldots, F_{m-1}$. For example, if $F(\vec{u}) = F_1(F_0(\vec{u}))$, then naively the prover could use Pinocchio twice to prove:

\begin{align*}
\vec{z} &= F_0(\vec{u}) \\
\vec{y} &= F_1(\vec{z})
\end{align*}

The verifier would check a proof for each equation separately and check that the output from $F_0$ was correctly used as input to $F_1$. Unfortunately, this means that the prover must send the intermediate state $\vec{z}$ to the verifier, and the verifier must perform work linear in $|\vec{z}|$. If $\vec{z}$ is large, then handling so much intermediate state would make it difficult or impossible for the verifier to benefit from outsourcing.

Instead, with Geppetto, the prover returns a constant-sized digest, $D_\vec{z}$, representing the intermediate state $\vec{z}$. The verifier uses this digest when checking the proof for Equation (5.0) and when checking the proof for Equation (5.1), ensuring that the prover consistently used the same intermediate state in both proofs, but without requiring the verifier to explicitly handle $\vec{z}$.

Prior work achieved a similar reduction in verifier effort by extending $F_0$ to hash its output and $F_1$ to hash its input, so the verifier need only handle the constant-sized hash value [BEG+91, Mer89, GGPR12, BSCGT13]. However, those hash computations make both functions more expensive [BFR+13, BCTV14]. In contrast, with Geppetto, a key observation is that Pinocchio already computes a digest-like structure
and that, with a careful refinement of its encoding, we can have the prover compute digests almost for free.\footnote{We use ‘digest’ rather than ‘commitment’, since only some of the digests need to be binding—see §5.3.1.}

In more detail, Geppetto divides all of the variables used to compute $F$ into disjoint sets called \textit{banks}. Each bank falls into one of three categories: a bank may represent $F$’s (the overall computation’s) input and output ($\vec{u}$ and $\vec{y}$ in the earlier example); it may represent a set of ‘local’ variables used within a single $F_i$; or it may be a \textit{bus}, i.e., a set of variables shared between multiple $F_i$ (e.g., $\vec{z}$).

Each bank is associated with its own cryptographic key material, used to compute a succinct digest of the values assigned to the bank’s variables: the prover produces a digest for each local bank and for each bus, while the verifier produces a digest for the IO banks as part of the verification process. The latter ensures that the proof verification is with respect to the input the verifier supplied, and the alleged output the prover produced.

To verify a proof that a given $F_i$ was computed correctly, the verification algorithm will need a digest for $F_i$’s local bank, and digests for any buses or IO banks that $F_i$ reads or writes.

Continuing with the earlier example, the verifier computes IO digests $D_u$ and $D_y$. The prover computes and returns digests $D_{F_0}$ and $D_{F_1}$ summarizing the intermediate variables used by $F_0$ and $F_1$ respectively, and a single digest $D_z$ representing the values on the bus between them. He also returns proofs $\pi_0$ and $\pi_1$ to demonstrate that $F_0$ and $F_1$ were computed correctly. The verifier runs the verification algorithm twice:

$$\text{Verify}((D_u, D_{F_0}, D_z), \pi_0) \quad (5.2)$$
$$\text{Verify}((D_z, D_{F_1}, D_y), \pi_1) \quad (5.3)$$

and accepts $y$ as $F(u)$ if both checks succeed.\footnote{This approach generalizes Pinocchio’s, which calls $(D_{F_0}, \pi_0)$ the proof for $F_0$ and has the verifier compute $D_u$ and $D_z$ inside the verification algorithm.} Note that $D_z$ occurs in both verification checks. Formally, a system that allows a prover to commit to state in this fashion and use the resulting commitments in multiple proofs is known as a commit-and-prove (CP) scheme (see §5.3.1).

As shown in Figure 5.1, proofs of complex functions $F$ may involve multiple instances of a simpler function $F_i$. For example, $F_i$ may represent the execution of a single function call, or a single loop iteration in $F$. Each instance of $F_i$ requires the prover to generate (and the verifier to check) a fresh proof, along with digests for the banks involved. Section 5.1.1 presents a formalization of these relationships with a \textit{proof schedule} (Defn 1); each step in the schedule indicates which $F_i$ is “active”, which banks it depends on, and which set of bank values this particular instance of $F_i$ depends on.

To efficiently build a commit-and-prove system supporting such schedules, Geppetto uses Pinocchio’s techniques to express each function $F_i$ as a Quadratic Arithmetic Program (QAP) $Q_i$, a format suitable for succinct cryptographic proofs. To share state between individual $Q_i$, they are combined into a single \textit{MultiQAP} $Q^\ast$ that also efficiently incorporates the buses connecting them. Using a MultiQAP also simplifies some definitions, constructions, and security proofs. In particular, it is possible to repeatedly use a commit-and-prove scheme for a single relation for all proof schedules composed of different $Q_i$ steps, with the ability to share compact, private digests between the proof steps. MultiQAPs support this functionality without significantly increasing the prover’s costs beyond what is required to handle each sub-QAP of the schedule individually.
A proof schedule (Defn. 1) with length
Commit-and-prove message for bank
Formal variables used when computing
The MultiQAP
A partition of
An instance
A QAP has size $F$

As described in §5.1.1, Geppetto decomposes the proof of a complex function $\sigma$ (5.0) and (5.1) would be
This encoding enables Pinocchio’s efficient cryptographic protocol.

To understand Geppetto’s MultiQAPs, it helps to review how Pinocchio encodes computations as QAPs. This encoding enables Pinocchio’s efficient cryptographic protocol.

Figure 5.2: Notation summary for §5.1.

<table>
<thead>
<tr>
<th>$F; F_0, \ldots, F_{m-1}$</th>
<th>Function $F$ is decomposed into $m$ functions $F_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\chi$</td>
<td>Formal variables used when computing $F$</td>
</tr>
<tr>
<td>$\vec{B}, \ell$</td>
<td>A partition of $\chi$ into banks $B_b \in \vec{B}$ with $\ell \triangleq</td>
</tr>
<tr>
<td>$B^{(t_b)}_b$</td>
<td>An instance $t_b$ of bank $B_b$;</td>
</tr>
<tr>
<td>$\chi_b$</td>
<td>Commit-and-prove message for bank $B_b$ (Defn. 2)</td>
</tr>
<tr>
<td>$\sigma, n$</td>
<td>A proof schedule (Defn. 1) with length $n \triangleq</td>
</tr>
<tr>
<td>$Q^*, Q_i$</td>
<td>The MultiQAP $Q^*$, combining sub-QAPs $Q_i$</td>
</tr>
<tr>
<td>$\rho, d$</td>
<td>A QAP has size $\rho$ and degree $d$</td>
</tr>
</tbody>
</table>

**Scheduling Proofs With Shared State**

As described in §5.1.1, Geppetto decomposes the proof of a complex function $F$ into a conjunction of proofs of $m$ simpler functions $F_0, \ldots, F_{m-1}$.

Let $\chi$ represent all of the formal variables used when computing $F$; this includes $F$’s input and output variables, variables “local” to the computation of each $F_i$, and the variables shared across the $F_i$. Based on these different roles, one can partition $\chi$ into banks $B_b \in \vec{B}$.

A given execution of $F$ may involve several instances of the same bank (e.g., if $F_i$ represents a loop body, then the banks corresponding to its IO and local variables may take on different concrete values on each loop iteration). These distinct instances of bank $B_b$ are denoted $B^{(t_b)}_b$ for $t_b = 1, 2, \ldots$ reserving $t_b = 0$ for the instance that assigns the constant 0 to every variable in $B_b$.

The notation described above will be used in the following definition.

**Definition 1** (Multi-proof schedule). A schedule $\sigma$ is a sequence of steps of the form $(i, \vec{t})$ where $i \in [m]$ and $\vec{t}$ is a vector with an index $t_b \geq 0$ for each bank $B_b \in \vec{B}$. Define $n \triangleq |\sigma|$ to be the length of the schedule, and $\ell \triangleq |\vec{B}|$ the number of banks.

Each step $(i, \vec{t})$ of the schedule selects a function $F_i$ and the instances $B^{(t_b)}_b$ of the banks it uses, with $t_b = 0$ whenever $F_i$ does not use $B_b$. $F_i$ is restricted to use only its local bank $B_{F_i}$, that is, $t_{F_j} = 0$ whenever $i \neq j$.

A proof for $\sigma$ consists of (1) a proof $\pi_i$ for each of its steps, and (2) a digest $D^{(t)}_b$ for each of its bank instances $B^{(t)}_b$.

Intuitively, the schedule indicates a sequence of calls to $F_i$s for which the prover must generate (or the verifier must check) a proof, and the indexes $\vec{t}$ of the banks digests that the prover (or the verifier) should use with that proof. The variables in any banks not used in a given step are implicitly set to 0 and hence can be represented with a trivial digest.

Returning to the example from §5.1.1, $\vec{B} = (B_u, B_y, B_{F_0}, B_{F_1}, B_z)$ and the schedule for Equations (5.0) and (5.1) would be $\sigma = [(0, (1, 0, 1, 0, 1)), (1, (0, 1, 0, 1, 1))]$.

**An Efficient CP System from MultiQAPs**

To understand Geppetto’s MultiQAPs, it helps to review how Pinocchio encodes computations as QAPs. This encoding enables Pinocchio’s efficient cryptographic protocol.

---

4Cryptographers think of $F$ as a language, and $F$’s IO as a language instance. Programmers may see this proof as a trace-property, e.g., interpreting $\vec{u}, \vec{y}$ as a valid input-output sequence obtained by running a program whose specification is captured by $F$. 

72
Quadratic Arithmetic Programs (QAPs) [PGHR13, GGPR13] Abstractly, Pinocchio compiles a function $F$ into a conjunction of $d$ equations of the form

$$Q(\chi) \triangleq \bigwedge_{r \in [d]} (\vec{v}_r \cdot \chi)(\vec{w}_r \cdot \chi) = (\vec{y}_r \cdot \chi)$$

(5.4)

where $\chi$ is the vector of $F$’s variables, which range over some large, fixed prime field $\mathbb{F}_p$, and the vectors $\vec{v}_r, \vec{w}_r, \vec{y}_r$ each define linear combinations over the variables $\chi$. Each equation (indexed by $r$) can be thought of as encoding a two-input multiplication gate in the arithmetic circuit computing $F$, with $\vec{v}_r$ indicating each variable’s contribution (if any) to the gate’s left input, $\vec{w}_r$ indicating each variable’s contribution to the gate’s right input, and $\vec{y}_r$ indicating the variable’s relation to the gate’s output.

$Q$ has size $\rho \triangleq |\chi|$ and degree $d$.

Crucially, Pinocchio’s evaluation key (used by the prover to create his proof) contains cryptographic key material for each variable $\chi \in \chi$, and the structure of that key material depends directly on which (and how) $\chi$ participates in each of the $d$ equations in Equation (5.4), i.e., on the value of $\chi$’s entry in each of the vectors $\vec{v}_r, \vec{w}_r, \vec{y}_r$.

From QAPs to MultiQAPs By decomposing $F$ into simpler functions $F_i$, it is possible create a corresponding QAP $Q_i$ for each $F_i$. Now consider the problem of connecting $Q_0$, which has some variables $\vec{z}_0$ representing $F_0$’s output, with $Q_1$, which has some variables $\vec{z}_1$ representing $F_1$’s input, with $|\vec{z}_0| = |\vec{z}_1|$. Since $F_0$ and $F_1$ are different functions, $\vec{z}_0$ and $\vec{z}_1$ undoubtedly participate in different equations in $Q_0$ and $Q_1$, and hence, as explained above, will have different key material representing $\vec{z}_0$ and $\vec{z}_1$. As a result, a digest for $\vec{z}_0$ will be completely different from a digest for $\vec{z}_1$, even if $\vec{z}_0 = \vec{z}_1$.

One possible approach to fixing this is combining $Q_0$ and $Q_1$ into a single QAP and adding equations requiring that $\vec{z}_0 = \vec{z}_1$, but that sacrifices the benefits of decomposing $F$.

Combining all of the $(Q_i)_{i \in [m]}$ into a single MultiQAP $Q^*$ is more promising. $Q^*$ has the same equations and variables $\chi$ used in the $Q_i$. In addition, for each variable $s$ that is shared between some subset $\vec{Q}$ of the $Q_i$, add a new variable $\hat{s}$ to a new bus bank associated with $\vec{Q}$, and add an equation relating $\hat{s}$ to the local copy of $s$ in each of the $Q_i$ in $\vec{Q}$. Continuing with the earlier example, add a new bus for variables $\vec{z}$ with $|\vec{z}| = |\vec{z}_0| = |\vec{z}_1|$, and for each $\hat{z}$ in $\vec{z}$, add an equation:

$$z_0 + z_1 = \hat{z}$$

(5.5)

relating it to the corresponding variables in $Q_0$ and $Q_1$. By adding the $\hat{z}$ bus as a layer of indirection, it no longer matters if $z_0$ is used differently in $Q_0$ than $z_1$ is in $Q_1$; the prover can create a single digest $D_{\vec{z}}$ representing the values on the bus, and the verifier can use this digest when checking the correct execution of $Q_0$, as well as that of $Q_1$, just as in the example in §5.1.1, when computing Equations (5.2) and (5.3). Because the verifier only accepts proof schedules with trivial digests for all other local banks (Definition 1), when she verifies a proof of $Q_0$, all of the variables in $Q_1$ are set to 0, and hence Equation (5.5) says that $z_0 = \hat{z}$, whereas when she verifies a proof of $Q_1$, all of the variables in $Q_0$ are set to 0, and hence (5.5) says that $z_1 = \hat{z}$.

Following these steps to combine $m$ sub-QAPs $(Q_i)_{i \in [m]}$, each of size $\rho_i$ and degree at most $d$, along with the buses connecting them, into a single MultiQAP $Q^*$, then $Q^*$ has size $\rho^* = |\vec{s}| + \sum_{i \in [m]} \rho_i$ and degree $d^* = d + |\vec{s}|$, where $\vec{s}$ includes all intermediate variables shared between the $Q_i$. By choosing a decomposition from $F$ to $(F_i)_{i \in [m]}$ that exploits the structure of $F$, Geppetto’s compiler can ensure that
most variables are local to one $F_i$, so typically $|\overline{s}| \ll d$. Since each step in a proof schedule considers only one $Q_i$ at a time, the size and degree of the “active” QAP within $Q^*$ is only slightly larger than the original $Q_i$. Thus, MultiQAPs enable state sharing across sub-QAPs without significantly increasing the prover’s costs beyond what is required to handle the sub-QAPs of the schedule individually.

5.2 Verifiable Crypto and Bootstrapping Proofs

In theory it should be possible to verify cryptographic computations (e.g., a signature verification) just like any other computation. In practice, as discussed above, a naive embedding of cryptographic computations into the field $\mathbb{F}_p$ that MultiQAPs operate over leads to significant overhead. As described in §5.5, the Geppetto implementation used a careful choice of cryptographic primitives and parameters to build a large class of crypto operations (e.g., signing, verification, encryption) using elliptic curves built “natively” on $\mathbb{F}_p$. This makes it cheap to verify computations on signed data, since the data and the signature both “live” in $\mathbb{F}_p$. Prior work used such tailoring for unbounded bootstrapping [BSCTV17] and hashing [BFR+13, BSCTV17].

The most complex application of this technology is a form of proof bootstrapping [Val08, BCCT13], which will address the main drawback of CP schemes. With CP schemes, including the MultiQAP-based scheme, the size of the cryptographic evidence—and the verifier costs—grow linearly with the number of digests and proofs. While often acceptable in practice, these costs can be reduced to a constant by using another instance of the CP scheme to outsource the verification of all of the cryptographic evidence according to a target proof schedule.

More formally, let $\text{Verify}_{\sigma^*}(\overline{D}, \Pi)$ be the function checking that a scheduled CP proof cryptographically verifies, where $\overline{D}$ and $\Pi$ are the collections of digests and proofs used in the schedule $\sigma^*$. Geppetto will be recursively applied to generate a quadratic program $Q_{\sigma^*}$ for $\text{Verify}_{\sigma^*}$. This yields another, more efficient verifier $\text{Verify}^*_{\sigma^*}(D^\sigma, \pi^\sigma)$ with a single, constant-sized digest $D^\sigma$ of $\overline{D}$, $\Pi$, and all intermediate variables used to verify them according to $\sigma^*$, and with a single constant-sized proof $\pi^\sigma$ to verify, now in constant time.

Furthermore, observe that $\text{Verify}^*_{\sigma^*}$ need not be limited to just verifying the execution of $\text{Verify}_{\sigma^*}$. For example, suppose an authority the client trusts (e.g., the US FDA) cryptographically signs the verification keys for $\text{Verify}_{\sigma^*}$, and define $\text{Verify}^*_{\sigma^*}$ to first verify the signature on the keys before using them to run $\text{Verify}_{\sigma^*}$. By using Geppetto’s option to make digests and proofs perfectly hiding, the verifier checks a constant-sized proof and learns that a trusted algorithm (for example, a medical diagnosis) ran correctly over her data, but she learns nothing about the algorithm. Thus, a client can efficiently and verifiably outsource computations with proprietary algorithms.

Although the general idea of bootstrapping is well-known [Val08, BCCT13], its practicality relies on careful cryptographic choices to support an efficient embedding. Recent work [BSCTV17] instantiated and implemented an embedding that supports bootstrapping an unbounded number of proofs but this generality comes at a cost. This is explained in more detail below in Section 5.5.

Using multiple levels reduces both the key sizes and the prover’s work. For example, suppose the application produces $N$ proofs for $\mathcal{P}$. The naïve approach of using a single recursive level $\mathcal{P}'$ would require a key capable of consuming all $N$ proofs. Instead, with multiple levels, we can design $\mathcal{P}'$ to consume $\sqrt{N}$ proofs from $\mathcal{P}$ and design $\mathcal{P}''$ to consume $\sqrt{N}$ proofs from $\mathcal{P}'$. The resulting keys will be $O(\sqrt{N})$, instead of size $N$ for a single recursive layer.
5.3 Defining Proof Composition

This section presents formal cryptographic definitions for the concepts introduced in §5.1. An exposition of the concrete protocol will follow in §5.4.

5.3.1 Commit-and-Prove Schemes

As discussed in §5.1.1, Geppetto employs three types of digest, one for F’s IO, one for the local variables for each $F_i$, and one for each bus. Each digest, $D$, may hide the values it represents via randomness $o$. Without hiding, we use a trivial opening $o = 0$ (and may omit it). We require that all digests of bus values be binding, as otherwise the prover could, say, use one set of values for the bus when proving that $F_0$ correctly wrote to the bus, while using a different set of values when proving that $F_1$ correctly read from the bus. In contrast, digests used only in a single proof, e.g., for intermediate local variables, need not be binding, since the verifier only needs to know that there exists an assignment of values to those variables corresponding to a single correct execution. Finally, digests of IO naturally need not be binding since the verifier computes them herself.

As a side note, while Geppetto uses commit-and-prove schemes to prove function executions, such schemes also enable interactive protocols where values are committed, used in proofs, and opened dynamically. For instance, they easily integrate with existing $\Sigma$-protocols as employed in anonymous credential systems [CL01, BL13].

Since we are interested in succinct proofs, we modify earlier definitions of commit-and-prove schemes [EG13, CLOS02, Kil89] to only consider computationally bounded adversaries. As a succinct digest implies that more than one plaintext maps to a given digest value, an unbounded adversary can always “escape” the digest’s binding property.

Each MultiQAP $Q^*$ in our construction defines a relation $R$ from the family $\mathcal{R}$ of all MultiQAPs over a fixed field $\mathbb{F}$. As our security definition has a security parameter $\lambda \in \mathbb{N}$ (which intuitively determines the size of the field $\mathbb{F}$), we actually talk about a sequence of families of polynomial-time verifiable relations $\{R_\lambda\}_{\lambda \in \mathbb{N}}$.

Definition 2 (Succinct Commit-and-Prove). Consider $\ell$-ary polynomial-time verifiable relations $\{R_\lambda\}_{\lambda \in \mathbb{N}}$ on tuples $\chi$ of a fixed length $\ell$.

A succinct commit-and-prove scheme $P = (\text{KeyGen} = (\text{KeyGen}_1, \text{KeyGen}_2), \text{Digest}, \text{Prove}, \text{Verify})$ for $\{R_\lambda\}_{\lambda \in \mathbb{N}}$ consists of five polynomial-time algorithms as follows:

- **Key generation is split into two probabilistic algorithms:**
  
  $\tau \leftarrow \text{KeyGen}_1(1^\lambda)$ takes the security parameter $\lambda$ as input and produces a trapdoor $\tau = (\tau_{\text{SIM}}, \tau_{\text{EXT}})$ (independent of $R$ and consisting of a simulation and extraction component).

  $(E_K, V_K) \leftarrow \text{KeyGen}_2(\tau, R)$ takes the trapdoor and a relation $R \in \mathcal{R}_\lambda$ as input and produces a public evaluation key $E_K$ and a public verification key $V_K$. To simplify notation, we assume that $E_K$ includes a copy of $V_K$, and that $E_K$ and $V_K$ include digest keys $E_K_b$ and $V_K_b$ for $b \in [\ell]$.

- **$D_b^{(t)} \leftarrow \text{Digest}(E_K_b, \chi_b^{(t)}, o_b^{(t)})$:** Given an evaluation key for $b$, message instance $t$ for $b (\chi_b^{(t)})$, and corresponding randomness $o_b^{(t)}$, the deterministic digest algorithm produces a digest $D_b^{(t)}$ of $\chi_b^{(t)}$. 

75
Proof-verification guarantees apply only when each digest $D_b^{(t)}$ in $\tilde{D}$ either passes the digest-verification algorithm or was computed directly by the verifier.

We define two security requirements below. Standard definitions for correctness and zero-knowledge are in the full paper [CFH+14]. First, we require that digests shared across multiple proofs (i.e., those representing bus values) be binding, meaning the prover cannot claim the digest represents one set of values in the first proof and a different set of values in the second proof. We collect the indexes of their keys in what we call the binding digest subset $S \subseteq [\ell]$.

**Definition 3 (Binding).** The commit-and-prove scheme $P$ is binding for $\ell$-ary relations $\{R_\lambda\}_{\lambda \in \mathbb{N}}$ and binding digest subset $S \subseteq [\ell]$, if for all efficient $A$ and any $R \in R_\lambda$,

\[
\Pr[ \tau \leftarrow \text{KeyGen}_1(1^\lambda); \tau = (\tau_{\text{SIM}}, \tau_{\text{EXT}}); \\
(EK, VK) \leftarrow \text{KeyGen}_2(\tau, R); \\
(b, \chi, o, \chi', o') \leftarrow A(EK, R, \tau_{\text{EXT}}); \\
\chi \neq \chi' \land b \in S \land \\
\text{Digest}(EK_b, \chi, o) = \text{Digest}(EK_b, \chi', o') ] = \text{negl} \lambda.
\]

Second, we require that if an adversary creates a set of digests and a proof that Verify accepts, then the adversary must “know” a valid witness, in the sense that this witness can be successfully extracted by “watching” the adversary’s execution. Note that the trapdoor the extractor receives from KeyGen$_1$ is generated independently of relation $R$ and hence cannot make it easier for the extractor to produce its own witnesses.

**Definition 4 (Knowledge Soundness).** The commit-and-prove scheme $P$ is knowledge sound for $\ell$-ary relations $\{R_\lambda\}_{\lambda \in \mathbb{N}}$, if for all efficient $A$ there is an efficient extractor $E$ taking the random tape of $A$ such that, for any $R \in R_\lambda$,

\[
\Pr[ \tau \leftarrow \text{KeyGen}_1(1^\lambda); \tau = (\tau_{\text{SIM}}, \tau_{\text{EXT}}); \\
(EK, VK) \leftarrow \text{KeyGen}_2(\tau, R); \\
(\tilde{D}, \pi; \chi, \tilde{\omega}) \leftarrow (A(EK, R) \parallel E(EK, R, \tau_{\text{EXT}})) : \\
(\exists b \in [\ell]. \text{Verify}(VK_b, D_b^{(t)}) \land D_b^{(t)} \neq \text{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})) \lor \\
(\forall b \in [\ell]. \text{Verify}(VK_b, D_b^{(t)} \land \text{Verify}(VK, \tilde{D}, \pi) \land \chi \notin R) ] = \text{negl} \lambda.
\]
5.3.2 Composition by Scheduling

As discussed in §5.1, intuitively, we can verify the correct execution of a complex function by verifying simpler functions and using digests to share state between them. We now formalize this intuition by extending knowledge soundness to multiple related proofs that share digests according to a proof schedule.

**Definition 5 (Scheduled Knowledge Soundness).** The commit-and-prove scheme \( \mathcal{P} \) is scheduled knowledge sound for \( \ell \)-ary relations \( \{ R_\lambda \}_{\lambda \in \mathbb{N}} \) and binding digest subset \( S \subseteq [\ell] \), if for all efficient \( \mathcal{A} \) there is an efficient extractor \( \mathcal{E} \) taking the random tape of \( \mathcal{A} \) such that, for any \( R \in R_\lambda \),

\[
\Pr[ \tau \leftarrow \text{KeyGen}_1(1^\lambda); \\
(\mathcal{E}K, \mathcal{V}K) \leftarrow \text{KeyGen}_2(\tau, R); \\
(\sigma, \mathcal{D}, \Pi; \chi, \tilde{\sigma}) \leftarrow (\mathcal{A}(\mathcal{E}K, R) \ || \ \mathcal{E}(\mathcal{E}K, R, \tau)) : \\
\forall D_{b}^{(t)} \in \mathcal{D}. (\text{Verify}(\mathcal{V}K_{b}, D_{b}^{(t)}) \Rightarrow D_{b}^{(t)} = \text{Digest}(\mathcal{E}K_{b}, \chi_{b}^{(t)}, o_{b}^{(t)})) \land \\
(\forall D_{b}^{(t)} \in \mathcal{D}. \text{Verify}(\mathcal{V}K_{b}, D_{b}^{(t)})) \land \\
\forall(i, \tilde{\sigma}) \in \sigma. \text{Verify}(\mathcal{V}K, \tilde{D}^{(t)}, \pi_{i})) \\
\Rightarrow \forall(i, \tilde{\sigma}) \in \sigma. \chi^{(t)} \in R \\
\] = 1 − \text{negl}(\lambda),

where \( \tilde{D}^{(t)} \) indicates a digest instance \( t \) for each bank \( b \) used in a given proof (and default digests of 0 values for any bank not used), and \( \chi^{(t)} \) represents the digested values.

**Theorem 2 (Scheduled Knowledge Soundness).** If a CP \( \mathcal{P} \) is knowledge sound and binding for \( \ell \)-ary relations and binding digest subset, then it is scheduled knowledge sound for the same relations and subset.

**Proof of Theorem 2 (Scheduled soundness):** Consider an adversary \( \mathcal{A} \) against the scheduled knowledge soundness of the proof system. \( \mathcal{A} \) takes \( \mathcal{E}K \) as input. We define \( n \) adversaries \( \mathcal{A}_1, \ldots, \mathcal{A}_n \) such that \( \mathcal{A}_i \) takes \( \mathcal{E}K \) as input and behaves like \( \mathcal{A} \) except that it discards all sub-proofs except \( \pi_i \). We construct our proof as a sequence of games.

Game 1 is the scheduled knowledge soundness game.

Game 2 is the same as Game 1, except that we run all of the extractors \( E_1, \ldots, E_n \) for \( \mathcal{A}_1, \ldots, \mathcal{A}_n \), whose existence is guaranteed by knowledge soundness, in parallel with \( \mathcal{A} \) and on the same input and random tape. Game 2 aborts without \( \mathcal{A} \) winning if for some \( \pi_i \), \( \text{Verify}(\mathcal{V}K, \tilde{C}^{(i)}, \pi_i) \) and all \( \text{Verify}(\mathcal{V}K_j, D_{b}^{(t)}) \) accept, but the \( E_i \) output witnesses \( \tilde{u} \) and randomness \( \tilde{\sigma} \) such that either \( \tilde{u} \in R \) or \( D_{b}^{(t)} \neq \text{Digest}(\mathcal{E}K_j, u_{j,t}, o_{j,t}) \) for some \( D_{b}^{(t)} \in \tilde{D} \).

**Lemma** The difference in the success probabilities of \( \mathcal{A} \) between Game 1 and Game 2 is negligible, based on the knowledge soundness of \( \mathcal{P} \).

Game 3 is the same as Game 2, except that it aborts without \( \mathcal{A} \) winning if for some \( i, i', \) and \( j \) \( \in S \), we have \( u_{j,t} \neq u'_{j,t} \). (Recall that all digests that appear twice in \( \sigma \) have to be digests and thus in \( S \) according to scheduled knowledge soundness.)
**Lemma** The difference in the success probabilities of $A$ between Game 2 and Game 3 is negligible by the binding property of the digest scheme.

The reduction runs the extractors and returns the collision to break the binding property. It relies on the adversary, and thus the reduction, receiving $\tau_{\text{EXT}}$ as input in the binding game.

In Game 3, $A$'s probability of success is 0, since for every proof $A$ returns, we can recover a witness such that $\bar{w}^{(i)} \in R$.

### 5.3.3 Combining Digests and Bootstrapping

As discussed in §5.2, the goal of proof bootstrapping [Val08, BCCT13] is to prove that $F$ was computed correctly using a succinct proof whose length does not depend on the length of the proof schedule. While Geppetto is compatible with both unbounded and bounded bootstrapping, our instantiation and implementation focuses on bounded bootstrapping, which we discuss below.

With bounded bootstrapping, given an initial CP scheme $\mathcal{P}$, we define a second CP scheme $\mathcal{P}'$ that verifies a fixed number $L$ of digests and proofs from $\mathcal{P}$. If our application generates more than $L$, we can define a third CP scheme $\mathcal{P}''$ that condenses digests and proofs from $\mathcal{P}'$. In theory, this process can be repeated recursively allowing for arbitrary many levels.

With enough levels, we can ensure that the verifier only receives a constant-sized digest and proof, and hence only performs work linear in the overall computation’s IO, regardless of how the prover decomposes $F$ into smaller functions.

Using multiple levels reduces both the key sizes and the prover’s work. For example, suppose the application produces $N$ proofs for $\mathcal{P}$. The naïve approach of using a single recursive level $\mathcal{P}'$ would require a key capable of consuming all $N$ proofs. Instead, with multiple levels, we can design $\mathcal{P}'$ to consume $\sqrt{N}$ proofs from $\mathcal{P}$ and design $\mathcal{P}''$ to consume $\sqrt{N}$ proofs from $\mathcal{P}'$. The resulting keys will be $O(\sqrt{N})$, instead of size $N$ for a single recursive layer.

As a scheduled CP may have many more digests than proofs, it is important for bootstrapping to also reduce the size of digests. We achieve this by partitioning digests into a smaller number of virtual banks. This is again a commit-and-prove scheme in which digests are represented by tuples of digests. In the bootstrapped CP, each of these partitions will be represented by a single compact digest.

For instance, we may consider two virtual banks and write $u_0$ for the subset of messages tagged as public, and $u_1$ for the others, minimizing the number of digests in the final proof. (Intuitively, $u_0$ includes messages passed in the clear and used to recompute digests.)

**Protocol 1** (CP Bootstrapping).

- **KeyGen**$(1^\lambda, R)$:
  
  $E_1, V_1 \leftarrow \text{KeyGen}(1^\lambda, R)$; $E_1', V_1' \leftarrow \text{KeyGen}(1^\lambda, R_{E_1})$ where
  
  $$(\bar{D}, \bar{\pi}) \in R_{E_1} \Leftrightarrow \bigwedge_{j=0}^{\ell-1} \text{Verify}(V_1, D_j) \land \text{Verify}(V_1, \bar{D}, \bar{\pi}).$$

  Let $E_1^o = (E_1', E_1) \text{ and } V_1^o = V_1'$ and output keys $E_1^o = (E_1, E_1')$ and $V_1^o = V_1'$. 

78
Theorem 3 (Bootstrapped CPs). If the CP schemes \( \mathcal{P} \) and \( \mathcal{P}' \) are knowledge-sound for \( \{R_\lambda\}_{\lambda \in \mathbb{N}} \) and \( \{R'_\lambda\}_{\lambda \in \mathbb{N}} \), respectively, then \( \mathcal{P}^0 \) as described in Protocol 1 is knowledge sound for \( \{R_\lambda\}_{\lambda \in \mathbb{N}} \).

Moreover, if \( \mathcal{P} \) and \( \mathcal{P}' \) are binding for some digests, then \( \mathcal{P}^0 \) is also binding for them, hence scheduled knowledge-sound. Finally, if \( \mathcal{P} \) and \( \mathcal{P}' \) are perfectly zero-knowledge then \( \mathcal{P}^0 \) is perfectly zero-knowledge.

This describes a single level of bootstrapping, but as the result is again a commit-and-prove scheme, unbounded boot-strapping is covered as well.

**Proof of Theorem 3 (CP bootstrapping soundness):** Consider an adversary \( \mathcal{A}^0 \) against the knowledge soundness of the constructed proof system. \( \mathcal{A}^0 \) takes \( EK^0 \) as input. We define an adversary \( \mathcal{A}' \) that takes \( EK' \) and \( R_{EK} \) as its input. \( \mathcal{A}' \) obtains \( EK \) from \( R_{EK} \) and behaves like \( \mathcal{A}^0 \) except that it moves \( D_\pi \) from the proof to the digests. We construct our proof as a sequence of games.

Game 1 is the original knowledge soundness game.

Game 2 is the same as Game 1, except that we run the extractors \( E' \) for \( \mathcal{A}' \), whose existence is guaranteed by knowledge soundness, in parallel with \( E^0 \) and on the same input and random tape as \( \mathcal{A}' \). Game 2 aborts without \( \mathcal{A}^0 \) winning if \( \text{Verify}(VK'_j, (\bar{D}, D_\pi), \pi') \) accepts, but \( E' \) does not output a witness \( \bar{D}, \pi \) such that \( (\bar{D}, \pi) \notin R_{EK} \).

**Lemma.** The difference in the success probabilities of \( \mathcal{A}^0 \) between Game 1 and Game 2 is negligible, based on the knowledge soundness of \( \mathcal{P}' \).

Let \( \text{EXT}^0 \) be the extractor built from both \( \text{EXT}' \) and \( \text{EXT} \) for \( \mathcal{P} \) and \( \mathcal{P}' \) respectively. In Game 2 the success probability of \( \mathcal{A}^0 \) is negligible, as \( R_{EK} \) guarantees that \( \text{Verify}(VK, \bar{D}, \pi) \) accepts, and because \( \mathcal{P} \) is a knowledge sound proof system for the relation \( R_\lambda \).

In the bootstrapping theorem, unlike in prior work [CKLM13], we avoid the controversial use of an auxiliary input [BCPR14]. The missing information for running \( \mathcal{A}' \) is obtained from the relation \( R_{EK} \) being proven.
5.4 Geppetto’s CP Protocol

This section presents a construction of an efficient commit-and-prove protocol for ℓ-ary relations \( \{ R_{Q^i} \}_{\lambda \in \mathbb{N}} \) (see §5.3.1) defined by a MultiQAP \( Q^* \) derived from multiple QAPs \( Q_\ell \), as described in §5.1.1.

5.4.1 MultiQAPs as Polynomials

Geppetto uses Pinocchio’s technique (which originated with Gennaro et al. [GGPR13]) to lift quadratic programs to polynomials.

Given MultiQAP \( Q^* \), of size \( \rho^* \) and degree \( d^* \), first define a set \( D \) of \( d^* \) “root values” of the form \( r \in \{ 2^i \}_{i=1}^{d^*} \) and define the polynomial \( \delta(x) \) as the polynomial with all \( r \in D \) as roots.

Recalling §5.1.1, define a set \( \mathcal{V} \) of \( \rho^* \) polynomials \( v_k(x) \) by interpolation over the roots in \( D \) such that for \( k \in [\rho^*], r \in D: v_k(r) = \overline{v}_{r,k} \). Each of the \( k \) polynomials essentially summarizes the effect one of \( \chi \)'s variables has on the computation. Define similar sets \( \mathcal{W} \) and \( \mathcal{Y} \) using the vectors \( \overline{w}_r \) and \( \overline{y}_r \).

The polynomial MultiQAP is satisfied by \( \overline{\chi} \) if \( \delta(x) \) divides \( p(x) \), where:

\[
p(x) = \left( \sum_{k=0}^{\rho^*} \chi_k \cdot v_k(x) \right) \cdot \left( \sum_{k=0}^{\rho^*} \chi_k \cdot w_k(x) \right) - \left( \sum_{k=0}^{\rho^*} \chi_k \cdot y_k(x) \right).
\]

Geppetto uses MultiQAPs to prove statements about shared state. To achieve this, the polynomials corresponding to bus values need to fulfill an additional condition. A bus bank \( B_b \) is commitment compatible if (i) the polynomials in each set \( \{ y_k(x) \}_{k \in B_b} \) are linearly independent, meaning that no linear combination of them cancels all coefficients, and (ii) all polynomials in the set \( \{ v_k(x), w_k(x) \}_{k \in B_b} \) are 0.

The first property is crucial for commitments to be binding, while the second improves performance and facilitates zero-knowledge when using externally generated commitments.

By inspection of Equation (5.5), the buses in the MultiQAP construction in §5.1.1 are commitment compatible. Concretely, continuing the example from that section, Equation (5.5) will be encoded as the QAP equation:

\[
(0 + \cdots + 0)(0 + \cdots + 0) = (1 \cdot z_0 + 1 \cdot z_1 + (1) \cdot \hat{z}).
\]

5.4.2 Commit-and-Prove Scheme for MultiQAPs

Geppetto’s protocol inherits techniques from Pinocchio [PGHR13]; the key differences are starting with MultiQAPs instead of QAPs, and splitting the prover’s efforts into separate digest and proof computations.

The following presentation of the protocol is given in terms of a generic quadratic encoding \( E \) [GGPR13]. In the implementation, the encoding is based on bilinear groups. Specifically, let \( e \) be a non-trivial bilinear map \( BF01 \) \( e : G_1 \times G_2 \rightarrow G_T \) and let \( g_1, g_2 \) be generators of \( G_1 \) and \( G_2 \) respectively. To simplify notation, define the encoding \( E(x) \) to be either \( g_1^x \) or \( g_2^x \) depending on whether it appears on the left or the right side of a product *.

Below, each \( B_b \subseteq B \) represents a subset of \( [\rho^*] \), and the commit-and-prove message \( \chi_b(t) \) represents the values of bank instance \( B_b(t) \).

**Protocol 2 (Geppetto).**

---

\[5\] Choosing roots of this form enables the C++ library to implement an efficient \( d^* \log d^* \) algorithm [BS05b] for the prover’s polynomial division.
• \( \tau \leftarrow \text{KeyGen}_1(1^\lambda) \):

Choose \( s, \{ \alpha_{v,b}, \alpha_{w,b}, \alpha_{y,b} \}_{b \in \mathbb{F}} \), \( r_v, r_w \overset{R}{\leftarrow} \mathbb{F} \). Construct \( \tau \) as \( (\tau_{\text{SIM}}, \tau_{\text{EXT}}) = (s, \{ \alpha_{v,b}, \alpha_{w,b}, \alpha_{y,b} \}_{b \in \mathbb{F}}, r_v, r_w) \).

• \((EK, VK) \leftarrow \text{KeyGen}_2(\tau, RQ^*)\):

Choose \( \{ \gamma_b, \beta_b \}_{b \in [\ell]} \overset{R}{\leftarrow} \mathbb{F} \). Set \( r_y = r_v \cdot r_w \).

To simplify notation, define \( E_v(x) = E(r_v x) \) (and similarly for \( E_w \) and \( E_y \)).

For the MultiQAP \( Q^* = (\rho^*, d^*, B, \mathcal{V}, \mathcal{W}, \mathcal{Y}, \delta(x)) \), construct the public evaluation key \( EK \) as:

\[
(EK_b)_{b \in [\ell]} \quad \left( E(s^i) \right)_{i \in [d]} \quad E_v(\delta(s)) \quad E_w(\delta(s)) \quad E_y(\delta(s))
\]

where each bank’s digest key \( EK_b \) is defined as:

\[
\begin{align*}
E_v(v_k(s)) & , \quad E_w(w_k(s)) & , \quad E_y(y_k(s)) \\
E_v(\alpha_{v,b}v_k(s)) & , \quad E_w(\alpha_{w,b}w_k(s)) & , \quad E_y(\alpha_{y,b}y_k(s)) \\
E(\beta_b(r_v v_k(s) + r_w w_k(s) + r_y y_k(s))) & , \quad \{ k \in B_b \}
\end{align*}
\]

\[
E_v(\alpha_{v,b}\delta(s)) \quad E_w(\alpha_{w,b}\delta(s)) \quad E_y(\alpha_{y,b}\delta(s)) \\
E_v(\beta_b\delta(s)) \quad E_w(\beta_b\delta(s)) \quad E_y(\beta_b\delta(s)).
\]

Construct the public verification key \( VK \) as:

\[
(VK_b)_{b \in [\ell]} \quad \left( E(1) \right) \quad E_y(\delta(s))
\]

where each bank’s digest verification key \( VK_b \) is:

\[
VK_b = E(\alpha_{v,b}) \quad E(\alpha_{w,b}) \quad E(\alpha_{y,b}) \quad E(\gamma_b) \quad E(\beta_b \gamma_b)
\]

Additionally \( VK \) includes digest keys \( EK_b \) for digests that the verifier computes (e.g., for IO banks).

Since \( EK \) and \( VK \) are public, the split into prover and verifier keys is primarily designed to reduce the verifier’s storage overhead.

• \( D_b^{(t)} \leftarrow \text{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)}) \):

Parse \( o_b^{(t)} \) as \( (\alpha_v, \alpha_w, \alpha_y) \).

If \( B_b \) is an IO bank, simply return:

\[
E_v(v^{(b)}(s)) \quad E_w(w^{(b)}(s)) \quad E_y(y^{(b)}(s))
\]

where \( v^{(b)}(s) = \sum_{k \in B_b} \chi_k v_k(s) + \alpha_v \delta(s) \) (and similarly for \( w^{(b)}(s) \) and \( y^{(b)}(s) \)). Since the verifier typically computes these digests, \( \alpha_v \) is typically 0.

Note that all of these terms can be computed using the values in \( VK_b \), thanks to the linear homomorphism of the encoding \( E \).

For any other bank, compute:

\[
E_v(v^{(b)}(s)) \quad E_w(w^{(b)}(s)) \quad E_y(y^{(b)}(s)) \\
E_v(\alpha_{v,b}v^{(b)}(s)) \quad E_w(\alpha_{w,b}w^{(b)}(s)) \quad E_y(\alpha_{y,b}y^{(b)}(s)) \\
E(\beta_b(r_v v^{(b)}(s) + r_w w^{(b)}(s) + r_y y^{(b)}(s)))
\]

81
Note that all of these terms can be computed using the values in $EK$. The values above constitute an extractable digest of the $\chi^{(t)}$ values, perfectly hidden via $o^{(t)}_b$. For commitment-compatible bases, this digest is also binding. Furthermore, for all commitment-compatible bases, $v^{(b)}(s), w^{(b)}(s), o_v, o_w$ are all 0, so the digest above simplifies to:

$$E_y(y^{(b)}(s)), E_y(\alpha_y, b^{(b)}(s)), E(\beta(b) y^{(b)}(s)) ;$$

- $\pi \leftarrow \text{Prove}(EK, \tilde{\chi}, \tilde{o})$: Parse each $o_b \in \tilde{o}$ as $(o_{b,v}, o_{b,w}, o_{b,y})$ and use the coefficients $\tilde{\chi}$ to calculate:

$$v(x) = \sum_{k \in [\rho^*]} \chi_k v_k(x) + \sum_{b \in [t]} o_{b,v} \delta(x),$$

and similarly for $w(x)$, and $y(x)$.

Just as in a standard QAP proof [GGPR13], calculate $h(x)$ such that $h(x)\delta(x) = v(x)w(x) - y(x)$, 
that is, the polynomial that proves that $\delta(x)$ divides $v(x)w(x) - y(x)$. Compute the proof as $\pi \leftarrow E(h(s))$ using the $E(s^i)$ terms in $EK$.

- $\{0, 1\} \leftarrow \text{Verify}(VK, D^{(t)}_b)$: Verify digest $D^{(t)}_b$ by checking

$$E_v(v^{(b)}(s)) * E(\alpha_v, b^{(b)}(s)) = E_v(\alpha_{v,b} v^{(b)}(s)) * E(1) \quad (5.6)$$

$$E_w(w^{(b)}(s)) * E(\alpha_w, b^{(b)}(s)) = E_w(\alpha_{w,b} w^{(b)}(s)) * E(1) \quad (5.7)$$

$$E_y(y^{(b)}(s)) * E(\alpha_y, b^{(b)}(s)) = E_y(\alpha_{y,b} y^{(b)}(s)) * E(1) \quad (5.8)$$

and the $\beta$ check:

$$E \left( \beta(b) v^{(b)}(s) + r_w w^{(b)}(s) + r_y y^{(b)}(s) \right) * E(\gamma_b) =$$

$$E(v^{(b)}(s)) + E(y^{(b)}(s))) * E(\beta(b)\gamma_b) + E(\beta(b)\gamma_b) * E(w^{(b)}(s)). \quad (5.9)$$

(For bases, the checks in Equations (5.6) and (5.7) are not necessary, and it is possible simplify the $\beta$ check (Eqn (5.9)).)

- $\{0, 1\} \leftarrow \text{Verify}(VK, D_0, \ldots, D_{t-1}, \pi)$: Combine the digests and perform the divisibility check on the proof term $E(h(s))$ in $\pi$:

$$\left( \sum_{b \in [t]} E_v(v^{(b)}(s)) \right) * \left( \sum_{b \in [t]} E_w(w^{(b)}(s)) \right) \quad (5.10)$$

$$- \left( \sum_{b \in [t]} E_y(y^{(b)}(s)) \right) * E(1) = E(h(s)) * E(\delta(s)) .$$

As described, the protocol supports non-interactive zero-knowledge proofs, in addition to verifiable computation. For applications that only desire the latter, the multiples of $\delta(s)$ in the $EK$ and the use of digest randomizations $o$ may be omitted.

**Theorem 4.** Protocol 2 has binding digests, as defined by Definition 3 under the $d$-SDH assumption.

**Proof of Theorem 4 (Binding):** We use the $E(s^i)$ values given as input to the $d$-SDH assumption to generate $EK$ and $VK$. An adversary that breaks the binding property produces $u = (c_k)_{k \in I}, o, u' = (c'_k)_{k \in I}, o', u \neq u'$, such that $\varphi(s) = \sum_{k \in I} c_k y_k(s) + o d(s) - \sum_{k \in I} c'_k y_k(s) - d' d(s) = 0$, i.e. $s$ is a root of $\varphi(s)$, and by factoring $\varphi(x)$, a reduction can easily find $s$ and thus break $d$-SDH. \qed
Theorem 5. Protocol 2 is a knowledge sound commit-and-prove scheme, as defined by Definition 4.

Proof of Theorem 5 (Knowledge Soundness): Consider an efficient adversary \( A \) who succeeds in the knowledge-soundness game in Definition 4. Given \( A \), we need to show that there exists an algorithm \( \mathcal{E} \) that, given the same randomness, input, and auxiliary information as \( A \) and the additional trapdoor \( \tau_{\text{ext}} \), produces the witnesses and openings \( A \) uses in its commitments and proof.

We will show that \( \mathcal{E} \) exists if extractors \( \chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell] \) exist. We do this by constructing \( \mathcal{E} \) from these constituent extractors. They in turn exist under the \( d \)-PKE assumption, whenever \( A \) exists, as we can construct the corresponding adversaries \( A_{v,j}, A_{w,j}, \) and \( A_{y,j} \) from \( A \).

To give more detail, we show how to construct \( A_{v,j}, j \in [\ell] \) from \( A \). Suppose \( A \) outputs commitments \( C_0, \ldots, C_{\ell-1} \) and proof \( \pi = H \), where each \( C_j = (V_j, V_j', W_j, W_j', Y_j, Y_j', Z_j), j \in [\ell] \). Each adversary \( A_{v,j} \) takes \( \{E(s^i)\}_{i \in [0..d]}, \{E(\alpha_{v,j}s^i)\}_{i \in [0..d]} \) as its main input and \( (E_K \setminus \{(E_{v}(v_k(s)) \cup E_{v}(\alpha_{v,j}v_k(s)))\}_{k \in I_j}, r_v, V_K \setminus \{E(\alpha_{v,j})\}) \) as its \( z \) input. Note that the assumption \( \alpha = \alpha_{v,j} \), and that \( z \) is independent of \( \alpha \) and thus computable by the state generator \( S \). \( A_{v,j} \) runs \( A \) after extending \( z \) to the full \( E_K \) and \( V_K \) by using its main input and \( r_v \) to recomputing \( \{(E_{v}(v_k(s)) \cup E_{v}(\alpha_{v,j}v_k(s)))\}_{k \in I_j} \), but outputs only \( (V_{j}, V_{j}') \). We construct adversaries \( A_{w,j} \) and \( A_{y,j} \) in a similar fashion.

Since verification succeeds, we know that \( V_j \ast E_v(\alpha_{v,j}) = V_{j}' \ast E_v(1) \). Under the \( d \)-PKE assumption, we thus know that for every \( A_{v,j} \) there exists an extractor \( \chi_{v,j} \) (taking the same inputs) that produces the coefficients of a polynomial \( V_j(x) \), such that \( E_v(V_j(s)) = V_j. \) And similarly with respect to \( W_j(x) \) for \( A_{w,j} \) and \( Y_j(x) \) for \( A_{y,j} \).

Given the constituent extractors \( \chi_{v,j}, \chi_{w,j}, \chi_{y,j} \) \( \mathcal{E} \) computes \( \{E(\alpha_{v,j}s^i)\}_{i \in [0..d]}, \{E(\alpha_{w,j}s^i)\}_{i \in [0..d]} \), and \( \{E(\alpha_{y,j}s^i)\}_{i \in [0..d]} \) using \( r_v \) and \( r_w \) and splits \( E_K \) into \( \sigma \) and \( z \) to call them on their correct inputs.

We now consider several games to bound the success probability of the main extractor \( \mathcal{E} \).

- **Game 1** is the same as the original knowledge soundness game, except that we abort without \( A \) winning if for verifying commitments \( \chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell] \) do not succeed in producing polynomials \( V_j(x) \) such that \( V_j \neq E_v(V_j(s)) \) (and similarly for \( W_j(x) \) and \( Y_j(x) \)).

**Lemma 2.** The difference in the success probability of \( A \) between the original knowledge soundness game and Game 1 is bounded (via a union bound) by the sum of the failure probabilities of \( \chi_{v,j}, \chi_{w,j}, \chi_{y,j}, j \in [\ell] \).

- **Game 2** is like Game 1, except that it sets \( H(x) = (V(x)W(x) - Y(x))/d(x), \) where \( V(x) = \sum_{j \in [\ell]} V_j(x) \), where \( V_j(x) \) is the polynomial extracted by \( \chi_{v,j} \) (and similarly for \( W(x) \) and \( Y(x) \)). Game 2 aborts without \( A \) winning if \( H(x) \) has a non-trivial denominator.

**Lemma 3.** The difference in the success probability of \( A \) between Game 1 and Game 2 is bounded by the success probability of an attacker \( B_1 \) breaking the \( 2q \)-SDH assumption, and is thus negligible.

- **Game 3** is like Game 2, except that it aborts without \( A \) winning, if one of the commitment polynomial triples \( R_j(x) = (V_j'(x), W_j'(x), Y_j'(x)) \), where \( V_j'(x) = V_j(x) \) mod \( d(x) \) (and similarly for \( W_j' \) and \( Y_j' \)), is not in the linear subspace \( S_j \), generated by the polynomial triples \( \{(v_k(x), w_k(x), y_k(x))\}_{k \in I_j} \), where the linear operations are done element-wise.

**Lemma 4.** The difference in the success probability of \( A \) between Game 2 and Game 3 is bounded by the success probability of an attacker \( B_2 \) breaking the \( q \)-PDH assumption, and is thus negligible.
In Game 3, $A$ has zero success probability.

For this, we show that the aborts in Game 2 and 3 are the only two cases in which the proof verifies but the extracted witness is invalid. We assume that neither case 1 nor case 2 holds; i.e., $H(x)$ has no non-trivial denominator, and each of the $R_j(x)$ is in the linear subspace $S_j$. We will show that $V(x)' = V(x) \mod d(x)$, $W'(x) = W(x) \mod d(x)$, and $Y'(x) = Y(x) \mod d(x)$ are a MultiQAP solution; i.e., they can be written as linear combinations of $\{v_k(x)_{k \in [0..\rho]}\}$ and $\{w_k(x)_{k \in [0..\rho]}\}$ using the same coefficients $c$ as required by the commit-and-prove relation $R^*$, and $V'(x)W'(x) - Y'(x)$ is divisible by $d(x)$.

Since for each $j$, $(V_j'(x), W_j'(x), Y_j'(x))$, is in the linear subspace, generated by the tuples $\{(v_k(x), w_k(x), y_k(x))\}_{k \in I_j}$, we can write $V_j'(x) = \sum_{k \in I_j} c_kv_k(x)$, $W_j'(x) = \sum_{k \in I_j} c_kw_k(x)$, $Y_j'(x) = \sum_{k \in I_j} c_ ky_k(x)$. We thus have that $V'(x)$, $W'(x)$, and $Y'(x)$ indeed can be written as the same linear combination $\{c_k\}_{k \in [0..\rho]}$ of their polynomial sets, as required in a MultiQAP.

As $V_j'(x) = V_j(x) \mod d(x)$, we also have that $V_j(x) = V_j'(x) + o_{j,v}d(x)$ for some $o_{j,v}$, and similarly for $W_j(x)$ and $Y_j(x)$. Therefore, $V(x) = \sum_{j \in [\ell], k \in I_j} c_kv_k(x) + \sum_j o_{j,v}d(x)$, $W(x) = \sum_{j \in [\ell], k \in I_j} c_kw_k(x) + \sum_j o_{j,w}d(x)$, and $Y(x) = \sum_{j \in [\ell], k \in I_j} c_ky_k(x) + \sum_j o_{j,y}d(x)$. Since $H(x)$ has no nontrivial denominator, it follows that $d(x)$ evenly divides $V(x)W(x) - Y(x)$ and thus also $V'(x)W'(x) - Y'(x)$.

The commitment opening values are defined as $o_j = (o_{v,j}, o_{w,j}, o_{y,j})$, for $j \in [\ell]$. Hence $V'(x)$, $W'(x)$, and $Y'(x)$ constitute a MultiQAP solution $\tilde{c}$ for the commit-and-prove relation $R$, and hence valid witness and openings $u_0, \ldots, u_{\ell-1}, o_0, \ldots, o_{\ell-1}$.

\section{5.5 Verifiable Crypto Computations}

**Background** Pinocchio, along with the systems built atop it, instantiates its cryptographic protocol using pairing-friendly elliptic curves. Such curves ensure good performance and compact keys and proofs. An elliptic curve $E$ defines a group of prime order $p'$ where each element in the group is an $(x, y)$ point, with $x$ and $y$ drawn from a second field $\mathbb{F}_p$ of large prime characteristic $p$. When Pinocchio is instantiated with such a curve, the QAPs (and hence all verifiable computations) are defined over $\mathbb{F}_{p'}$, and hence code that compiles naturally to operations on $\mathbb{F}_{p'}$ is cheap.

**Approach** At a high-level, for Geppetto the curve $E$ is such that the group order “naturally supports” operations on a second curve $\tilde{E}$, which can be used for any cryptographic scheme built on $\tilde{E}$, e.g., anything from signing with ECDSA to the latest attribute-based encryption scheme.

In more detail, suppose Geppetto is used to verify ECDSA signatures over an elliptic curve $\tilde{E}$ built from points chosen from $\mathbb{F}_q$. By using a pairing-friendly elliptic curve $E$ with a group of prime order $p' = q$, then operations on points from $\tilde{E}$ embed naturally into QAPs, meaning that basic operations like adding two points cost only a handful of cryptographic operations, rather than hundreds or thousands required if $p'$ did not align with $q$.

**Bootstrapping** As described in §5.2, proof bootstrapping is a particularly compelling example of verifying cryptographic operations, since it allows us to condense a long series of proofs and digests into a single proof and digest.
Remarkably, Karabina and Teske [KT08] show that it is possible to generate two MNT curves [MNT01] $E$ and $\tilde{E}$ that are pairing friendly and, more importantly, $\tilde{E}$ can be embedded in $E$, and $E$ can be embedded in $\tilde{E}$.

Ben-Sasson et al. [BSCTV17] recently instantiated and implemented such curves to bootstrap the verification of individual CPU instructions. Geppetto can use a similar approach to achieve unbounded bootstrapping of entire QAPs. Specifically, we could instantiate two versions of Geppetto, one built on $E$ that condenses proofs consisting of points from $\tilde{E}$ and another built on $\tilde{E}$ that condenses proofs consisting of points from $E$.

Unfortunately, there are drawbacks to using the curves Ben-Sasson et al. found. First, they were only able to find a pair of curves that provide 80 bits of security. Finding cycles of performant curves for the more standard 128-bit setting appears non-trivial, since just finding 80-bit curves required over 610,000 core-hours of computation. Second, the MNT curve family is not the most efficient family at higher security levels, and achieving a cycle requires larger-than-usual fields, creating additional inefficiency [BSCTV17].

Geppetto instead uses a sequence of nested curves (an option suggested previously [BSCTV17, Footnote 10]) to instantiate and implement bounded bootstrapping as a pragmatic alternative. Specifically, we instantiate one version of Geppetto with the same highly efficient BN curve [BN06] employed by Pinocchio. We use the BN curve to generate a collection of digests and proofs for our MultiQAP-based CP scheme. We then construct a second curve capable of efficiently embedding the BN curve operations. When instantiated with the second curve, Geppetto can efficiently verify crypto operations on the BN curve. Thus, a verifier can, for example, check signatures on the verification key built on the BN curve and then use that key to verify the BN digests and proofs. To gain greater scalability, this process can be repeated with a bounded number of additional carefully constructed curves, each used to verify the digests and proofs from the previous curve. Unfortunately, none of the curves can efficiently embed later curves, and hence when generating keys, the client must ultimately commit to the maximum number of BN proofs that will be verified.

Details We construct bilinear systems, $G_{IN}$ and $G_{OUT}$. To achieve this at the 128-bit security level, we instantiate $G_{IN}$ using a Barreto-Naehrig (BN) elliptic curve [BN06], and then construct $G_{OUT}$ accordingly with the Cocks-Pinch method [CP01]. Roughly, the latter constructs a pairing-friendly curve by outputting a finite field corresponding to a given, prescribed group order. We fix the prime $p$ from the BN parameterization as the group order, so that the output of the Cocks-Pinch algorithm is the prime $\tilde{p}$ (as well as the other parameters required in the description of $G_{OUT}$). The following lemma makes this explicit in a special case that is of most interest in the current work.

**Lemma 5.** Let $x \in \mathbb{Z}$ be such that $p = 36x^4 + 36x^3 + 24x^2 + 6x + 1$ and $p' = 36x^4 + 36x^3 + 18x^2 + 6x + 1$ are prime. If

$$\tilde{p} = 5184x^8 + 10368x^7 + 12204x^6 + 8856x^5 + 4536x^4 + 1548x^3 + 363x^2 + 48x + 4$$

(5.11)

is also prime, then there exists both an elliptic curve $E/\mathbb{F}_p$ of order $\#E(\mathbb{F}_p) = p'$ with embedding degree $k = 12$ (with respect to $p'$), and an elliptic curve $\tilde{E}/\mathbb{F}_{\tilde{p}}$, such that its order $\#\tilde{E}(\mathbb{F}_{\tilde{p}})$ is a multiple of $p$ and $\tilde{E}$ has embedding degree $\tilde{k} = 6$ (w.r.t. $p$).
A full proof and details on this construction appear in the full version of the Geppetto paper [CFH+14].

To construct additional nesting curves, given a group order, we once again apply the Cocks-Pinch approach to produce a sequence of curves $E(i)$, defined over prime fields $F_{p_i}$, respectively, such that $p_i$ divides $\#E((i+1)(F_{p_{i+1}}))$. Each hop creates a larger curve, and hence will eventually produce curves equal to or larger than the MNT curves that support unbounded bootstrapping. For example, for the first Cocks-Pinch curve, $\tilde{p}$ is 509 bits (with embedding degree 6), and the next two levels are 1023 bits and 2055 bits with embedding degrees 3 and 1.

Even when we reach these larger sizes, the inner layers (especially the BN curve where most of the “real” computation happens) are still more efficient than the MNT curves, and even at comparable sizes, exponentiations on the Cocks-Pinch curves are faster due to a CM endomorphism (not available for MNT curves) and a $G_2$ cubic twist. Of course, for sufficiently large problems, the unbounded approach eventually offers better performance.

### 5.6 Evaluation

#### 5.6.1 Microbenchmarks

To calibrate our results, we summarize the cost of our cryptographic primitives in Figure 5.3. We generally use a Barreto-Naehrig (BN) curve for generating digests and proofs, and we use the Cocks-Pinch (CP) curves to handle embedded cryptographic computations like bootstrapping. We show measurements from two CP curves to illustrate how the costs grow for each progressive level. The BN curve is asymmetric, meaning that one source group (base) is cheaper than the other (twist). Geppetto’s protocol and compiler are designed to keep most of the work on the base group.

The CP curves are slower than the BN for two reasons. First, the CP curves are chosen to support bounded bootstrapping, so they use larger field elements than the BN curve (see §5.5). Second, the BN code has been extensively optimized, including hand-tuned assembly code, while the CP code is newly written C. Based on operation counts from Magma [BCP97], the first CP curve should be within 2-4 $\times$ of the BN curve, and indeed comparing the CP curve’s performance with a similar C version of the BN curve confirms this.

#### 5.6.2 MultiQAPs

The following experimental results compare the use of MultiQAPs for shared state with the use of hashing in prior work such as Pantry [BFR+13]. At a micro level, Pantry’s results suggest that hashing an element
of state increases the degree of the QAP by $\sim 11.25$/byte. In contrast, with MultiQAPs, a full field element only increases the degree by one, so with Geppetto, the degree only increases by $\sim 0.03$/byte, a savings of $375 \times$. Even if the operations are on 32-bit values, instead of full field elements, Geppetto only costs $0.25$/byte, a savings of $45 \times$.

At a macro level, for MapReduce, Pantry and Geppetto share the same costs for proving that the core mapper and reducer computations were performed correctly; on top of that, to handle state transferred between mappers and reducers, Pantry proves the correctness of $2M \cdot R$ hashes (since both the mappers and the reducers must prove they hashed the state correctly), while Geppetto proves the correctness of $M \cdot R$ bus digests. As a result, Geppetto’s keys end up being a bit smaller; Geppetto’s keys save further relative to Pantry, as Pantry needs key material for $R$ hashes for each mapper and $M$ hashes for each reducer, while Geppetto only needs $\max(M, R)$ shared buses.

A naïve alternative to MultiQAPs and hashing is to build one gigantic Pinocchio QAP, so that the shared state becomes simply internal circuit wires. However, experiments quickly showed the futility of this approach; even for the relatively modest applications shown in Figure 5.4 and assuming only 10 mappers, this approach would require a QAP with a degree of 10M+, while the Pinocchio prover keels over (i.e., begins swapping) before it can reach 3M on a 16 GB machine.

### Applications

To measure the end-to-end effect of MultiQAPs, Geppetto was evaluated using the following applications. Geppetto’s results were compared against Pantry’s implementation running on the same hardware, except where Pantry runs out of memory, in which case Pantry’s validated cost model [BFR+13] is reported.

The first two examples are borrowed from Pantry [BFR+13] to give a direct comparison with their work. The experimental setup adopted Pantry’s ratio of 10 mappers to 1 reducer, and uses their extension of Pinocchio to ensure an apples-to-apples comparison.

#### MapReduce: Dot Product [BFR+13]

The verifier specifies (in Pantry via hash, in Geppetto via a digest) two vectors of integers; each mapper receives $m$ integers and computes a partial dot product, and the reducer sums the mapper outputs.
MapReduce: Nucleotide Substring Search [BFR+13] The verifier specifies a DNA string that is divided amongst the mappers, each receiving \( m \) nucleotides. The mapper then searches for dynamically supplied length-\( d \) substrings reporting a match (if any) to the reducer which combines the matches.

Loop: Matrix Exponentiation The verifier supplies a dynamically chosen \( n \times n \) matrix \( M \) and an exponent \( e \), and the prover returns \( M^e \). Matrix exponentiation is useful for many applications, e.g., to compute the width of a graph represented as an adjacency matrix [Tha13].

This example shows the benefits of intertwined MultiQAPs. With Pinocchio, the QAP would scale with \( e \), limiting the size of the problem, whereas using MultiQAPs there is only a need to compile the loop body (after some loop unfolding), which can then be used for arbitrary values of \( e \). With Pantry, the loop body needs to hash a matrix on the way in and again on the way out, whereas MultiQAPs incur a handful of crypto operations per intermediate state generated.

MultiQAP Results

Figure 5.4 summarizes the impact of using MultiQAPs for shared state. The results only show CPU costs and do not include network latency or bandwidth, though the latter is unlikely to be a problem for either Pantry or Geppetto, given that proofs and digests are only a few hundred bytes each.

For MapReduce, note see the largest discrepancy between Geppetto and Pantry on the dot-product app. For this app, the QAP for the computation itself is quite simple, so for Pantry, the cost of hashing dwarfs the cost of the computation. For the nucleotide app, the shared state is still a dominant portion of the calculation for Pantry (though not as dominant as in dot product), and hence Geppetto maintains a wide margin.

For the Loop application, the QAP for the computation itself is non-trivial and grows faster than the IO between loop iterations; thus, the cost of state sharing relative to the computation is lower than for dot product, and the ratio drops further for larger matrices. Since Geppetto and Pantry generate essentially the same QAP for the computation itself, Geppetto’s relative advantage drops accordingly.

5.6.3 Verifying Cryptography and Bootstrapping

In §5.5, we claimed that embedding cryptographic operations without matching field sizes was exorbitantly expensive. To validate this claim, we combined data from a basic ECC pairing operation coded in Magma with cost models from Pinocchio for various operations such as bit splitting. Our calculations estimate that the pairing alone would require a QAP with degree of 44 million.

Fortunately, our choice of matching curves in §5.5 brings this cost down significantly. For example, a pairing only requires a QAP of degree 14K, an improvement of 3100× vs. the naïve approach, while an exponentiation, i.e., \( g^x \), increases the degree by \( \sim 60 \) per bit in \( x \).

Furthermore, as discussed in §5.5, for a comparable security level, our initial curves for bounded bootstrapping provide approximately 34-77× better performance than curves supporting unbounded bootstrapping [BSCTV17]. As §5.6.1 shows, however, performance degrades with each level added, and hence will eventually reach a point where they fall short of the unbounded curves’ performance.
Bootstrapping

From the verifier’s perspective, one level of bootstrapping is attractive, since she only receives (and only verifies) one constant-sized, 512-bit proof, and one constant-sized, 448-byte digest.

Without bootstrapping, the only way for the prover to generate such concise proofs would be via one massive Pinocchio-style QAP, which our results above (§5.6.2) show is infeasible. Nonetheless, bootstrapping does come at a cost. While bootstrapping, the “outer” QAP’s degree grows with each digest or proof that it must verify. We summarize these costs below assuming that the verification keys are known at compile-time.

- For each recomputed digest, we increase the degree by 2K for each 32-bit integer value committed.
- For each full digest verification, we pay 79.6K (including the pairings needed for the checks from Equations (5.6)-(5.9)).
- For each bus digest verification, we pay 33.8K (since, as noted in §5.4.2, buses require fewer checks).
- For each proof verification (Eqn (5.10)), we pay 28.2K.

With keys unknown at compile-time, we pay instead 89.8K and 30.6K for full digest and proof verification, respectively.

We also observe that the prover’s cryptographic cost for “outer” proofs and digests is typically higher than for work on the “inner” instance, even for QAPs of the same size. One reason is that the outer CP curve is less efficient than the inner BN curve (§5.6.1). A second reason is that many of the values the prover commits to for the inner instance arise from the program being verified, and hence they are often 1, 32, or 64 bits. In contrast, the outer curve verifies elliptic curve operations and hence many values are full-fledged 254-bit values.

While these costs are substantial, they are low enough that we can employ bootstrapping to scale the prover to much larger computations. For example, with our existing implementation, we could bootstrap up to 14 “inner” proofs sharing 16 buses; applying this to, say, the matrix exponentiation example allows us to produce a single, constant-size proof for a computation with a useful (i.e., not counting bootstrapping costs) QAP degree of over 50 M. When evaluating the computation, the prover executes 24M LLVM instructions and generates a proof in 152 minutes. While slow, this is five orders of magnitude faster than the unbounded bootstrapping in previous work (BCTV) [BSCTV17], which, with a reported clock rate of 26 milliHz (and a lower 80-bit security level), would take approximately 29 years.

No source code was available for BCTV, so analyzing the causes of this large performance gap requires some guesswork. First, we estimate that one order of magnitude comes from the different choices of curves.

Second, BCTV use a circuit that checks a general-purpose CPU transition function for each program instruction. Thus, for straight-line code like matrix multiplication, they use hundreds of equations for each operation, whereas Geppetto generally uses one. BCTV’s interpreter, however, supports RAM access and data-dependent control flow, while Geppetto’s compilation-based implementation currently does not, and thus, one might expect a smaller performance gap on applications making use of those features. However, recent work [WSR+15] indicates that the compilation-based approach can incorporate these features
and still outperform interpretation by 2-4 orders of magnitude on straight line code, and 1-3 orders of magnitude on RAM and data-dependent benchmarks.

Finally, BCTV apply bootstrapping at a very fine granularity. At every step of their CPU, they produce a proof with one curve, and then they use their second curve to verify that proof and translate it back into a proof on the first curve. Thus, each CPU instruction requires two bootstrapped proof verifications, whereas in this application, each Geppetto proof verification covers 1.7M LLVM instructions.

5.7 Chapter Acknowledgements

This chapter presents joint work with Bryan Parno, Jon Howell, Craig Costello, Cédric Fournet, Michael Naehrig, and Samee Zahur. The authors gratefully thank Joppe Bos, Olga Ohrimenko, Srinath Setty, Michael Walfish, and the anonymous reviewers. This work was done as part of an internship at Microsoft Research.
Part II

Protocols Supporting Concrete Real-World Applications
Chapter 6

Privately Measuring Advertisement Effectiveness

6.1 Introduction

This chapter describes a family of protocols that have been deployed to support the business needs of a large technology company. In this case, the company sells an advertisement targeting service based on data it has collected about consumers. The company’s targeting algorithm is intended to show advertisements to people based on their particular interests. The company wishes to keep information about which ads were shown to whom secret, as such information can reveal sensitive details of a person’s life.

On the other hand, the company’s customers wish to know the overall effectiveness of their advertising. In this case, “effectiveness” is the proportion of revenue from purchases made by people who saw an ad. This cannot be inferred from sales data alone; it is possible that during a period of increasing sales an advertising campaign actually has a negative impact i.e. that the sales might have increased even more had there been no advertising at all. The computation requires both sales data and knowledge of which customers saw an ad.

This is an instance of the Private Intersection-Sum problem [IKN+17], a variant of Private Set Intersection (PSI) where one or both of the parties’ input sets includes an associated value for each element, with the output being the sum over those associated values for the intersection. While it is certainly possible to compute this functionality using a generic protocol, in practice this would be too expensive, particularly in terms of the communication required. This motivates the need for a custom protocol for this application.

Two such protocols are presented in this chapter; these protocols has been deployed by a large tech company with a prominent targeted advertising business. Many of the technical decisions presented below are motivated by the real-world security and performance needs of the business applications. One of the most important constraints is the need to minimize communication cost, even if doing so increases the total running time; this rules out not just generic protocols but even state-of-the-art custom PSI protocols [RR17c, RR17b, CLR17], some of which are faster but require more communication.
6.1.1 MPC in Real-World Applications Perspective

With the growing number of Internet applications involving sensitive data held by multiple parties there has been growing demand for MPC applications, and several such applications have been previously reported. One of the first was the widely-cited Danish Sugar Beet Auction [BCD+09a]. In Estonia MPC is used for another financial application [BTW11]. Another recent application of MPC was reported in a survey of faculty salaries at universities in and around Boston, MA [LVB+16]. MPC was also combined with differential privacy techniques to gather statistics about how the anonymity network Tor is used [JJ16].

One distinguishing characteristic of the particular application described in this chapter over those previous applications is the need to regularly run large numbers of protocol instances per day. While the application described by Lindell et al. [AFL+16, FLNW16] has a similar requirement, it is run between machines located in the same data center and owned by a single organization. The advertising effectiveness application must run between two different organizations, using machines that cannot be physically co-located.

6.1.2 Chapter Organization

Section 6.2, gives a description of the basic Intersection-Sum protocol, with a detailed security analysis in the honest-but-curious model in Section 6.2.1. The “reverse” variant of the protocol, designed to satisfy certain externally-imposed requirements, is presented in Section 6.3. Related work is presented in Section 6.5.

6.2 Protocol Description

A detailed description of the Intersection Sum protocol is found in Figure 6.2. In the protocol presented, there are two participating parties, of which only Party 1 learns the cardinality of the intersection, and only Party 2 learns the intersection-sum.

At a high-level, the two parties interact to hash and “double-encrypt” each entry in their datasets, and compare the double-encrypted values. The “double-encryption” is similar to the deterministic Pohlig-Hellman cipher [HP84].

The group \( G \) can be any group in which the DDH assumption is believed to hold. Several candidate groups are widely used, such as subgroups of the multiplication group of a finite field and elliptic and hyperelliptic curve groups. In practice, carefully chosen elliptic curves like Curve25519 [Ber06] offer a good balance between security and performance.

In the security arguments the Random Oracle Model will be invoked. In this model, parties in the real world have access to an oracle RO that computes a random function over \( \{0, 1\}^* \). The simulator will simulate this oracle, choosing outputs for each oracle query made by the party under simulation. In a concrete implementation RO will be instantiated with a hash function such as SHA-3. Below, RO is assumed to output elements of \( G \); SHA-3 can be adapted to this case using rejection sampling, or alternatively \( G \) may be instantiated using Curve25519 so that all 256 bit strings are valid elements of the group. As is standard, the parties will prepend a different session identifier to the inputs to SHA-3 in each session to simulate the requirement that RO be different in each session.
6.2.1 Security Analysis

As discussed above, the security proof will be in the honest-but-curious model. This requires some degree of trust between the two parties not to deviate from the prescribed protocol. In practice this has been accepted by the users of the protocol, which are willing to rely on legal consequences to prevent malicious behavior.

The following proof is similar to the proof given by Agrawal et al. [AES03]. Security is shown by giving a simulator that can indistinguishably simulate the view of each honest party in the protocol given only that party’s input, the cardinality of the intersection, and the intersection-sum (but not the input of the other party). Intuitively, this will show that each party learns nothing more by participating in the protocol than the cardinality of the intersection and the intersection sum.

Let $\text{REAL}^\lambda(u_i \in [m], \{(v_j, t_j)\}_{j \in [n]})$ be a random variable representing the view of Party $i$ in a real protocol execution, where the random variable ranges over the internal randomness of all parties, and the randomness in the setup phase (including that of the Random Oracle).

Theorem 6 shows that Party 1’s view in the protocol can be simulated given only that Party 1’s input and the size of the intersection (but not the input of Party 1).

**Theorem 6 (Honest But Curious Security, against Party 1).** There exists a PPT simulator $\text{SIM}_1$ such that for all security parameters $\lambda$ and inputs $\{u_i\}_{i \in [m]}, \{(v_j, t_j)\}_{j \in [n]}$,

$$\text{REAL}^\lambda(u_i \in [m], \{(v_j, t_j)\}_{j \in [n]}) \approx \text{SIM}_1^\lambda(u_i \in [m], n, |J|)$$

Where $n$ is the size of Party 2’s input, $J = \{j : v_j \in \{u_i\}_{i \in [m]}\}$ is the intersection set, and $|J|$ is its cardinality.

**Proof.** The simulator algorithm $\text{SIM}_1$ is presented in Algorithm 3.

Notice that the main difference between $\text{SIM}_1$ and a real protocol execution is in Round 2: instead of sending $\{\text{RO}(u_i)^{k_1 k_2}\}$ and $\{\text{RO}(v_j)^{k_2}\}$ as in a real execution, $\text{SIM}_1$ instead uses random group elements $\{g_i\}$ and $\{h_j\}$ which have an intersection of the same size, and Paillier encryptions of 0.

$$\text{REAL}^\lambda(u_i \in [m], \{(v_j, t_j)\}_{j \in [n]}) \approx \text{SIM}_1^\lambda(u_i \in [m], |J|)$$

follows from a multi-step hybrid argument, where each neighboring pair of hybrid distributions is computationally indistinguishable.
Private Intersection-Sum Protocol

- **Setup:**
  - Both parties agree on a security parameter \( \lambda \) and a \( \mathcal{G} \in \mathcal{G}(\lambda) \), and a user identifier space \( \mathcal{U} = \mathcal{U}(\lambda) \). Both parties have access to a Random Oracle \( \text{RO} : \mathcal{U} \rightarrow \mathcal{G} \) that maps user identifiers to random elements of \( \mathcal{G} \).
  - Party 1 has as input a set \( U_1 = \{ u_i \}_{i \in [m]} \) of \( m \) user identifiers, where each \( u_i \in \mathcal{U} \).
  - Party 2 has as input a set \( \{ (v_j, t_j) \}_{j \in [n]} \) of \( n \) user identifiers paired with transaction values, where each \( v_j \in \mathcal{U} \), and each \( t_j \in \mathbb{Z}^\ast \), such that \( \sum t_j \) fits comfortably into the Paillier message space for security parameter \( \lambda \). We define \( U_2 = \{ v_j \}_{j \in [n]} \).
  - Each Party \( i \) chooses a random private exponent \( k_i \) in the group \( \mathcal{G} \).
  - Party 2 generates a fresh key-pair \( \text{Pk}(\lambda) \leftarrow \text{Paillier.Gen}(\lambda) \) and shares the public key \( \text{Pk} \) with Party 1.

- **Round 1 (Party 1):**
  1. For each element \( u_i \) in its set, Party 1 applies the Random Oracle and then single-encrypts them using its key \( k_1 \), thus computing \( \text{RO}(u_i)^{k_1} \).
  2. Party 1 sends \( \{ \text{RO}(u_i)^{k_1} \}_{i \in [m]} \) to Party 2 in shuffled order.

- **Round 2 (Party 2):**
  1. For each element \( \text{RO}(u_i)^{k_1} \) received from Party 1 in the previous step, Party 2 double-encrypts them using its key \( k_2 \), computing \( \text{RO}(u_i)^{k_1 k_2} \).
  2. Party 2 sends \( Z = \{ \text{RO}(u_i)^{k_1 k_2}, v_j \}_{i \in [m]} \) to Party 1 in shuffled order.
  3. For each item \( (v_j, t_j) \) in its input set, Party 2 applies the Random Oracle to the first element of the pair and encrypts it using key \( k_2 \). It encrypts the second element of the pair using the Paillier key \( \text{Pk} \). It thus computes the pair \( \text{RO}(v_j)^{k_2}, \text{Paillier}(t_j) \).
  4. Party 2 sends the set \( \{ \text{RO}(v_j)^{k_2}, \text{Paillier}(t_j) \}_{i \in [n]} \) to Party 1 in shuffled order.

- **Round 3 (Party 1):**
  1. For each item \( \{ \text{RO}(v_j)^{k_2}, \text{Paillier}(t_j) \} \) received from Party 2 in Round 2 Step 4, Party 1 double-encrypts the first member of the pair using \( k_1 \), thus computing \( \text{RO}(v_j)^{k_1 k_2}, \text{Paillier}(t_j) \).
  2. Party 1 computes the intersection set \( J \):
     \[ J = \{ j : \text{RO}(v_j)^{k_1 k_2} \in Z \} \]
     where \( Z \) is the set received from Party 1 in Round 1.
  3. For all items in the intersection, Party 1 homomorphically adds the associated ciphertexts, and computes a ciphertext encrypting the intersection-sum \( S_J \):
     \[ \text{Paillier}(\text{Pk}, S_J) = \text{Paillier.Sum}(\{ \text{Paillier}(t_j) \}_{j \in J}) = \text{Paillier} \left( \sum_{j \in J} t_j \right) \]
  4. Party 1 sends this ciphertext to Party 2.

- **Output (Party 2):** Party 2 decrypts the Paillier ciphertext received in Round 3 using the Paillier secret key \( sk \) to recover the intersection-sum \( S_J \).

Figure 6.2: Detailed description of the Private Intersection-Sum protocol.
Algorithm 3 The simulator for Party 1

Input: \( \{\lambda, \{u_i\}_{i \in [m]}, n, |J|\} \)  
Output: \( \text{SimView}(P_1) \) \( \text{SIM}_1 \lambda, \{u_i\}_{i \in [m]}, |J| \)

1: Generate key \( k_1 \in \mathcal{G} \), and Paillier key-pair \((pk, sk)\).
2: Honestly generate and send \( \{\text{RO}(u_i)_{k_1}\}_{i \in [m]} \) in shuffled order as Party 1’s message in Round 1.
3: Create a dummy set \( U^*_{P_2} = \{g_i\}_{i \in [m]} \), where each \( g_i \) is randomly selected from \( \mathcal{G} \). Send \( \{g^k_i\}_{i \in [m]} \) in shuffled order as Party 2’s message in Step 2 of Round 2.
4: Create a dummy set \( U^*_{P_2} = \{h_j\}_{j \in [n]} \) for Party 2 by setting \( h_j = g_j \) for \( j \in [1, |J|] \), and each \( h_j \) for \( j \in [|J|, m) \) is randomly selected from \( \mathcal{G} \).
5: Send \( \{(h_j, \text{Paillier}(pk, 0))\}_{j \in [n]} \) in shuffled order as Party 2’s message in Step 4 of Round 2, where each \( \text{Paill}() \) is freshly generated.
6: Honestly generate Party 1’s message in Round 3 using Party 2’s dummy messages from the previous step.
7: Output Party 1’s view in the simulated execution above.

Hyb\(_0\) The view of Party 1 in a real execution of the protocol.

Hyb\(_{1,0}\) The same as Hyb\(_0\), except, in Round 2, all Paillier ciphertexts sent by Party 2 are replaced with fresh encryptions of 0.

Hyb\(_{1,i}\) for \( i \in [m - |J|] \): The same as Hyb\(_{1,i-1}\), except with RO\((u_i)^{k_1}k_2\) replaced by \( g_i^{k_1} \) in Party 2’s message in Step 2 of Round 2, where \( u_i \) is the lexicographically smallest as-yet-unreplaced element of \( \{u_i\}_{i \in [m]} \setminus \{v_j\}_{j \in [n]} \) and \( g_i \) is a random element of \( \mathcal{G} \).

Hyb\(_{2,0}\) Identical to Hyb\(_{1,m-|J|}\).

Hyb\(_{2,j}\) for \( j \in [n - |J|] \): The same as Hyb\(_{2,j-1}\), except with RO\((v_j)^{k_2}\) replaced by \( h_j \) in Party 2’s message in Step 4 of Round 2, where \( v_j \) is the lexicographically smallest as-yet-unreplaced element of \( \{v_j\}_{j \in [n]} \setminus \{g_i\}_{i \in [m]} \) and \( h_j \) is a random element of \( \mathcal{G} \).

Hyb\(_{3,0}\) Identical to Hyb\(_{2,n-|J|}\).

Hyb\(_{3,k}\) for \( k \in [|J|] \): The same as Hyb\(_{3,k-1}\), except

- RO\((u_k)^{k_1}k_2\) replaced by \( g_k^{k_1} \) in Party 2’s message in Step 2 of Round 2
- RO\((v_k)^{k_2}\) replaced by \( g_k^{k_2} \) in Party 2’s message in Step 4 of Round 2

where \( u_k = v_k \) is the lexicographically smallest as-yet-unreplaced element of \( \{v_j\}_{j \in [n]} \cap \{g_i\}_{i \in [m]} \) and \( g_k \) is a random element of \( \mathcal{G} \).

Hyb\(_4\) The view of Party 1 output by \( \text{SIM}_1 \).

First observe that Hyb\(_0\) and Hyb\(_{1,0}\) are indistinguishable by the CPA security of the Paillier encryption scheme. Also observe that the pairs of hybrids (Hyb\(_{1,m-|J|}\), Hyb\(_{2,0}\)), (Hyb\(_{2,n-|J|}\), Hyb\(_{3,0}\)) and (Hyb\(_{3,|J|}\), Hyb\(_{4}\)) are identical.

It remains to show that hybrids of the form Hyb\(_{1,i-1}\), Hyb\(_{1,i}\), Hyb\(_{2,j-1}\), Hyb\(_{2,j}\) and Hyb\(_{3,k-1}\), Hyb\(_{3,k}\) are indistinguishable. Hyb\(_{1,i-1}\) and Hyb\(_{1,i}\) are indistinguishable for all \( i \in [m - |J|] \) based on the hardness. Note that hybrids of the form Hyb\(_{2,j-1}\), Hyb\(_{2,j}\) and Hyb\(_{3,k-1}\), Hyb\(_{3,k}\) can be proven indistinguishable by a very similar argument.
Consider Algorithm 4 below, that takes as input a DDH tuple \((g, g^a, g^b, g^c)\) and hybrid index \(i\), and simulates \(Hyb_{1,i}\):

**Algorithm 4** Simulator for \(Hyb_{1,i}\)

**Input:** \((\lambda, i, \{u_i, g, g^a, g^b, g^c\}, \{u_i\} \in [m], \{v_j\} \in [n])\)

**Output:** \(SimV iew(P_i)\) in \(Hyb_{1,i}\)

Sim:\(Hyb_{1,i}\) \((\lambda, i, \{g, g^a, g^b, g^c\}, \{u_i\} \in [m], \{v_j\} \in [n])\) with \(u_i\) being the newest element replaced with a random one in \(Hyb_{1,i}\)

1. \(for \ i \in [m] \ do\)
2. \(\quad if \ u_i \neq u_{i*} \ then\)
3. \(\quad \quad Randomly \ sample \ r_i \leftarrow [1, |G|]\)
4. \(\quad Program \ RO(u_i) = g^{r_i}\)
5. \(\quad else \ if \ u_i = u_{i*} \ then\)
6. \(\quad \quad Program \ RO(u_i) = g^a\)
7. \(\quad end \ if\)
8. \(end \ for\)
9. \(for \ j \in [n] \ do\)
10. \(\quad Randomly \ sample \ s_j \leftarrow [1, |G|]\)
11. \(\quad Program \ RO(v_j) = g^{s_j}\)
12. \(end \ for\)
13. \(Generate \ key \ k_1 \in G\), and Paillier key-pair \((pk, sk)\).
14. \(Send \ \{RO(u_i)^{k_1}\} \in [m] \ in \ shuffled \ order \ as \ Party \ 1’s \ message \ in \ Round \ 1.\)
15. \(for \ i \in [m] \ do\)
16. \(\quad if \ u_i = u_{i*} \ then\)
17. \(\quad \quad g_i \leftarrow g^c\)
18. \(\quad else \ if \ u_i \notin \{v_j\} \in [n], u_i < u_{i*} \ then\)
19. \(\quad \quad g_i \leftarrow \text{random} \ \text{element} \ \text{of} \ G\)
20. \(\quad else\)
21. \(\quad \quad g_i \leftarrow (g^b)^{s_i}\)
22. \(\quad end \ if\)
23. \(end \ for\)
24. \(Send \ \{g_i^{k_1}\} \in [m] \ in \ shuffled \ order \ as \ Party \ 2’s \ message \ in \ Step \ 2 \ of \ Round \ 2. \ Send \ \{(g^b)^{s_j}, \text{Paillier}(0)\} \in [n] \ in \ shuffled \ order \ as \ Party \ 2’s \ message \ in \ Step \ 4 \ of \ Round \ 2, \ where \ each \ Paillier(0) \ \text{is freshly generated.}\)
25. \(Honestly \ generate \ Party \ 1’s \ message \ in \ Round \ 3 \ using \ Party \ 2’s \ dummy \ messages \ from \ the \ previous \ step.\)
26. Output Party 1’s view in the simulated execution above.

Observe that the output distribution produced by Algorithm 4 on input \(i\) and a DDH tuple \((g, g^a, g^b, g^c)\) for uniformly random \(a, b, c\) is identical to \(Hyb_{1,i}\). To see this, first observe that the Random Oracle has uniformly random outputs even after reprogramming, since all the reprogrammed values are random powers of a generator. Next, interpreting the hidden exponent \(b\) as Party 2’s key \(k_2\), all the simulated messages sent by Party 2 in Round 2 are of the correct form for \(Hyb_{1,i}\): un-replaced messages in Round 2 Step 2 have the form \(RO(u_i)^{k_1}k_2\), and messages sent in Round 2 Step 4 have the form \((RO(v_j)^{k_2}, \text{Paillier}(0))\).

Next, replace the DDH tuple given as input to Algorithm 4 to have the form \((g, g^a, g^b, g^{ab})\). The only effect is that, instead of \(g_i = g^c\), we have \(g_i = g^{ab} = RO(u_i)^b\). From the earlier interpretation of \(b\) as \(k_2\), this means \(g_i^{k_2} = RO(u_i)^{k_1k_2}\). This change is exactly the difference between \(Hyb_{1,i-1}\) and \(Hyb_{1,i}\). Thus, the output of Algorithm 4 on inputs \(i\) and \((g, g^a, g^b, g^{ab})\) is identical to \(Hyb_{1,i-1}\).

From the preceding argument, it follows that if any adversary can distinguish between \(Hyb_{1,i-1}\) and
Hyb\textsubscript{1,i}, then it can distinguish between \((g, g^a, g^b, g^{ab})\) and \((g, g^a, g^b, g^{c})\). Therefore, by the assumed hardness of DDH, Hyb\textsubscript{1,i−1} and Hyb\textsubscript{1,i} are indistinguishable.

Theorem 7 shows that Party 2’s view in the protocol can be simulated given only that Party 2’s input and the intersection-sum (but not the input of Party 1).

**Theorem 7** (Honest But Curious Security, against Party 2). There exists a PPT simulator \(\text{SIM}\) such that for all security parameters \(\lambda\) and inputs \(\{u_i\}_{i \in [m]}, \{(v_j, t_j)\}_{j \in [n]}\),

\[
\text{REAL}^{2,\lambda}(\{u_i\}_{i \in [m]}, \{(v_j, t_j)\}_{j \in [n]}) 
\approx 
\text{SIM}_2(1^\lambda, \{(v_j, t_j)\}_{j \in [n]}, m, S_J)
\]

Where \(m\) is the size of Party 1’s input, \(J = \{j : v_j \in \{u_i\}_{i \in [m]}\}\) is the intersection set, and \(S_J = \sum_{j \in J} t_j\) is the intersection-sum.

**Proof.** Define \(\text{SIM}_2\) to perform the Setup phase honestly, and honestly performs the operations corresponding to Party 2. \(\text{SIM}_2\) simulates the messages sent by Party 1 as follows:

- In Round 1, instead of sending \(\{\text{RO}(u_i)^{k_1}\}_{i \in [m]}\) as Party 1’s message, \(\text{SIM}_2\) instead sends \(m\) randomly selected elements of \(G\).

- In Round 3, instead of performing the intersection and computing the intersection-sum, \(\text{SIM}_2\) instead sends a fresh Paillier ciphertext encrypting the value \(S_J\) it received as input.

Note that the only difference between the output of \(\text{SIM}_2\) and the view of Party 2 in a real execution is in the Round 1 messages. However, the Round 1 messages output by \(\text{SIM}_2\) can be shown to be indistinguishable from those in a real execution by using a simple hybrid argument: Define \(m\) hybrids, where, in each successive hybrid, \(\text{SIM}_2\) replaces one additional “real” Round 1 message of the form \(\text{RO}(u_i)^{k_1}\) with a random element of \(G\). Then, each pair of neighboring hybrids can be shown to be indistinguishable based on the fact that \(k_1\) is secret and that DDH is hard in \(G\). The details are very similar to the proof of Theorem 6.

### 6.2.2 Additional Security Precautions

The previous security analysis shows that each party learns no more than the size of the intersection and the intersection-sum. However, unless appropriate care is taken, these values may themselves leak private information. For example, if the intersection size is very small, it may be possible to guess the user identifiers in the intersection based on the intersection-sum. To guarantee enough mixing-privacy between the users, parties should ensure that the intersection is sufficiently large. The “reverse” protocol variant presented in Section 6.3 allows parties to enforce a minimum intersection size, by allowing them to abort before either party learns the intersection sum if the if the intersection is too small.

In general, though, privacy may be violated as a consequence of certain input distributions. For example, if there are “outlier” \(v_j\) values that are unusually large, the sum will be large; a priori knowledge of such values will allow a party to identify users. It is also possible that repeatedly executing the protocol in sequence will leak information due to correlated inputs in different sessions. Such problems are an
Figure 6.3: Summary of the Reverse Intersection-Sum protocol

artifact of the functionality itself and would affect any intersection-sum protocol. One strategy for resolving this issue would be to compose differential privacy techniques [DR14] with the cryptographic protocol, by adding appropriately sampled noise to the inputs.

### 6.3 Protocol Variant: The “Reverse” Protocol

The protocol presented in Section 6.2 can be modified in a straightforward way to allow both parties to learn the intersection-sum or intersection-size. It is also possible to ensure that one or the other party performs the actual intersection operation, for example, to allow that party to abort if the intersection is below some threshold, which might be imposed for policy reasons. One such variant is described in Figure 6.4, which is referred to as the “reverse” protocol. In this protocol, Party 2 performs the intersection, and can abort the protocol if the intersection size is too small, without either party learning the intersection-sum. In addition, both parties learn the intersection size, but only Party 1 learns the intersection-sum. To implement this, Party 1 additionally needs to blind the Paillier ciphertext with random values, as can be seen in Figure 6.4.

#### 6.3.1 Security Analysis

The security proof for the reverse protocol is similar to the ordinary protocol, but with the roles of the parties reversed, with Party 2 learning only the intersection size, and Party 1 learning both the intersection size and the intersection sum. The simulator for Party 1 is almost identical to the simulator for Party 2 in the original protocol, but must also provide indices $J$ in Round 3 to allow Party 1 to compute $S_j - \sum_{j \in J} r_j$. For Party 2, the simulator is very similar to the original Party 1 simulator $SIM_1$ in Algorithm 3. The details of this proof are omitted.

### 6.4 Reducing the Communication Overhead

In both the Forward and Reverse versions of the protocol we require a additive-homomorphic encryption scheme, and in the exposition in Sections 6.2 and 6.3, Paillier encryption was chosen for this purpose. Unfortunately, Paillier encryption results in a large ciphertext expansion if the values to be summed

---

1This leakage is implicit in the security proof. The simulator will receive the result of the function evaluated on both parties’ inputs; this result itself is subject to the problems above. The security proof shows that a party will learn only as much as the functionality reveals, which includes leakage due to the input distribution.
Reverse Intersection-Sum Protocol

- **Setup:**
  - Both parties agree on a security parameter $\lambda$ and a group $\mathcal{G} \in \mathcal{G}(\lambda)$, and a user identifier space $\mathcal{U} = \mathcal{U}(\lambda)$. Both parties have access to a Random Oracle $RO : \mathcal{U} \rightarrow \mathcal{G}$ that maps user identifiers to random elements of $\mathcal{G}$.
  - Party 1 has as input a set $\{u_i\}_{i \in [m]}$ of $m$ user identifiers, where each $u_i \in \mathcal{U}$.
  - Party 2 has as input a set $\{(v_j, t_j)\}_{j \in [n]}$ of $n$ user identifiers paired with transaction values, where each $v_j \in \mathcal{U}$, and each $t_j \in \mathbb{Z}^+$, such that $\sum t_j$ fits comfortably into the Paillier message space.
  - Each Party $i$ chooses a random private exponent $k_i$ in the group $\mathcal{G}$.
  - Party 2 generates a fresh key-pair $(pk, sk) \leftarrow \text{Pai.Gen}(\lambda)$ for the Paillier encryption scheme and shares the public key $pk$ with Party 1.

- **Round 1 (Party 2):**
  1. For each element $(v_j, t_j)$ in its set, Party 2 applies the Random Oracle and then single-encrypts $v_j$ using its key $k_2$, thus computing $RO(v_j)^{k_2}$.
  2. Party 2 sends $\{(RO(v_j)^{k_2}, \text{Pai}(t_j))\}_{j \in [n]}$ to Party 1 in shuffled order.

- **Round 2 (Party 1):**
  1. For each element $(RO(v_j)^{k_2}, \text{Pai}(t_j))$ received from Party 2 in the previous step, Party 1 double-encrypts them using its key $k_1$ and homomorphically computes a one-time pad encryption of $t_j$ under addition modulo the Paillier modulus $N$, computing $(RO(v_j)^{k_1k_2}, \text{Pai}(t_j + r_j))$.
  2. Party 1 sends $\{(RO(v_j)^{k_1k_2}, \text{Pai}(t_j + r_j))\}_{j \in [n]}$ to Party 2 in shuffled order. The (shuffled $j \rightarrow r_j$) map is saved for a future step.
  3. For each item $u_i$ in its input set, Party 1 applies the Random Oracle to the first element of the pair and encrypts it using key $k_1$. It encrypts the second element of the pair using the Paillier key $pk$. It thus computes the pair $RO(u_i)^{k_1}$.
  4. Party 1 sends the set $\{RO(u_i)^{k_1}\}_{i \in [m]}$ to Party 2 in shuffled order.

- **Round 3 (Party 2):**
  1. For each item $RO(u_i)^{k_1}$ received from Party 1 in Round 2 Step 4, Party 1 double-encrypts the using $k_2$, thus computing $RO(u_i)^{k_1k_2}$.
  2. Party 2 computes the intersection set $J$:
      \[ J = \{ j : RO(v_j)^{k_1k_2} \in \{RO(u_i)^{k_1k_2}\}_{i \in [m]} \} \]
  3. For all items in the intersection, Party 2 decrypts $\text{Pai}(t_j + r_j)$ and adds the associated (one-time pad encrypted) ciphertexts, computing a ciphertext encrypting the intersection-sum $S_J = \sum_{j \in J} t_j + r_j$.
  4. Party 2 sends $S_J$ together with the indexes $J$ corresponding to the Paillier ciphertexts in the intersection, to Party 1.

- **Output (Party 1):** Party 1 computes $S_J - \sum_{j \in J} r_j$ to recover $\sum_{j \in J} t_j$.

Figure 6.4: Detailed description of the “Reverse” Private Intersection-Sum protocol.
are relatively small, which increases the overall communication cost of the protocol. For example, a Paillier ciphertext with a relatively standard modulus of 2048 bits will have ciphertext size 4096 bits, with plaintext space 2048 bits. However, if the values to be summed are in the range of 20 bits, then using Paillier results in approximately $200 \times$ ciphertext expansion compared to the plaintext.

Several possible alternatives are discussed below.

### 6.4.1 Exponential ElGamal

If the spend values will typically be small, on the order of 20 bits, and the intersection is also expected to be small, e.g. around $2^{16}$, then exponential ElGamal is a good alternative candidate for homomorphic encryption. A typical choice of ElGamal parameters will result in ciphertexts of length 512 bits, which for a 20 bit value would be an expansion of $25 \times$. In the Forward protocol, after the sum the plaintext size will be 36 bits. Similarly, in the Reverse protocol, the masked sum would be computed in the exponent, which will be about 36 bits after the masks are subtracted.

This approach, trades communication for computation, as the decryption requires inverting a discrete logarithm; however, a trade-off between memory and CPU time can be made, if the range is small enough to permit a lookup table. For a 36 bit exponent the lookup table would be relatively large but well within the storage capacity of typical server-class machines.

### 6.4.2 Lattice-Based Cryptosystems

Another option is to use an lattice-based encryption scheme. Such schemes, which are based on the hardness of Learning-With-Errors or Ring-Learning-With-Errors, have ciphertexts with multiple ‘slots’, where each slot is individually additively homomorphic. Using such a scheme, one could encrypt multiple spend values into a single ciphertext by putting a value into each ciphertext slot. This would greatly reduce the ciphertext expansion, to about $1.5 \times$, using typical security parameters. This approach would work well in the Reverse protocol, where the homomorphism is used only to add masks to values, and the values in the intersection are decrypted before being summed.

One challenge with encrypting several values into a single ciphertext is that it become challenging to shuffle them. While lattice schemes can allow homomorphic shuffling of slots, this is expensive and requires a larger ciphertext expansion factor. An alternative is to do a block-wise shuffling, that is, to group IDs into blocks where each block’s values occupy a single LWE-ciphertext, and to send the blocks in a random order. This involves a security trade-off: since the IDs are not perfectly shuffled, some additional information about the intersection would leak as a result.

### 6.4.3 “Packing” Paillier

A final possibility is “packing” the plaintexts in the Paillier scheme, by splitting up the large Paillier plaintext space into slots, and encoding each plaintext value in a higher-order position from the previous value, leaving some additional “padding” bits for overflow bits of the sum. As with LWE-based approaches mentioned above, this would work best for the Reverse protocol, since the values are decrypted before being summed, allowing the vector to be unpacked prior to summing. Also as with LWE-based approaches, packing would involve a loss of security, since it is unclear how to shuffle the packed Paillier values.

These schemes are actually “fully”-homomorphic, but here we consider only their additive homomorphism.
Figure 6.5: Comparison of computation and communication required by the PSI protocol and previous work. For the DDH-based protocol we assume Curve25519 which has 32 byte group elements. For the Chen, Laine, and Rindal scheme, the byte count depends on the specific FHE parameters; their experiments were reported only for 32 bit strings, which makes a fair comparison in terms of bytes difficult.

However, with packing, Paillier (together with the Damgard-Jurik optimizations) can reach quite low ciphertext expansion. Among the optimizations, this can present the best communication overhead but requires the largest security trade-off.

6.5 Related Work

Private Set Intersection is a well-studied problem. The goal there is for both parties to learn the items in the intersection, but nothing more. There are many existing approaches in the literature, including works based on DDH-type assumptions [HFH99b, DCKT10, Lam16, SFF14], works based on Oblivious Transfer [PSZ14b, PSSZ15, DCW13b, RR16], works based on Oblivious Polynomial Evaluation [FNP04, DSMRY09] and works based on generic Secure Two-party Computation techniques [HEK12b, PSSZ15]. In the advertisement analysis setting the universe of possible set members is large, so techniques assuming bit-vector representations of the set are inapplicable.

Closer to this chapter’s goal, there are several works that limit the parties to learning only the cardinality of the intersection [FNP04, AES03, KS05, VC05, DCGT12, NAA+09]. Previous works using garbled circuits also allow cardinality, as well as more general functions of the intersection, however the communication overhead of garbled circuit approaches would be too high for this application. In Huang et al.’s work, the Sort-Compare-Shuffle approach, which is the most applicable to this setting, requires $O(n \log n)$ communication [HEK12b], and even with state-of-the-art garbling schemes will require far more communication than the DDH-based protocol here. Similarly, the Phasing technique of Pinkas et al. [PSSZ15] also requires quasi-linear communication.

Due to the “offline” nature of this use-case, which allows for higher latency, comparisons based on overall running time are less valuable than the concrete resource costs required to run the protocol. For this application the most limited resource was network capacity, so the comparisons are based on the amount of communication required to compute the result.
6.6 Chapter Acknowledgements

This chapter presents joint work with Moti Yung, Sarvar Patel, Karn Seth, Mihaela Ion, Ahmet Erhan Nergiz, Shobhit Saxena, and David Shanahan. This work was done at Google.
Chapter 7

Secure Aggregation of User Data

This chapter presents a protocol developed for a machine learning application for mobile phones, published by Bonawitz et al. [BIK + 17]. The core protocol has been described in previous work by Ács et al. [ÁC11], and is similar to another real-world application described by Jansen and Johnson [JJ16]. Unlike those previous protocols, the mobile device setting this protocol targets implies frequent devices failures. Dealing with these failures while remaining within application-imposed limits on communication is the main technical contribution of this chapter.

7.1 Introduction

Machine learning models trained on sensitive real-world data promise improvements to everything from medical screening [PWH16] to disease outbreak discovery [LMCS15]. And the widespread use of mobile devices means even richer—and more sensitive—data is becoming available [KMY + 16].

However, large-scale collection of sensitive data entails risks. A particularly high-profile example of the consequences of mishandling sensitive data occurred in 1988, when the video rental history of a nominee for the US Supreme Court was published without his consent [Adv87]. The law passed in response to that incident remains relevant today, limiting how online video streaming services can use their user data [McC12].

This work outlines an approach to advancing privacy-preserving machine learning by leveraging secure multiparty computation (MPC) to compute sums of model parameter updates from individual users’ devices in a secure manner. The problem of computing a multiparty sum where no party reveals its update in the clear—even to the aggregator—is referred to as Secure Aggregation. As described in Section 7.2, the secure aggregation primitive can be used to privately combine the outputs of local machine learning on user devices, in order to update a global model. Training models in this way offers tangible benefits—a user’s device can share an update knowing that the service provider will only see that update after it has been averaged with those of other users.

This chapter is particularly focused on the setting of mobile devices, where communication is extremely expensive, and dropouts are common. Given the expense of communication in this setting, one design goal is to ensure that the protocol requires no more than twice as much communication as sending the data vector to be aggregated in the clear, and would also like the protocol to be fully robust to users dropping at any point. Previous works do not address this mixture of constraints.
Figure 7.1: Left: In the cloud-centric approach to machine intelligence, user devices interact with cloud-hosted models, generating logs that can be used as training examples. The logs from many users are combined and used to improve the model, which is then used to serve future user requests. Middle: In Federated Learning, machine intelligence models are shipped to users’ devices where they are both evaluated and trained locally. Summaries of improved models are shared with the server, where they are aggregated into a new model and deployed to user devices. Right: When Secure Aggregation is added to Federated Learning, the aggregation of model updates is logically performed by the virtual, incorruptible third party induced by the secure multiparty communication, so that the cloud provider learns only the aggregated model update.

7.2 Federated Learning

Consider training a deep neural network to predict the next word that a user will type as she composes a text message. Such models are commonly used to improve typing efficacy for a phone’s on-screen keyboard [GVSP02]. A modeler may wish to train such a model on all text messages across a large population of users. However, text messages frequently contain sensitive information; users may be reluctant to upload a copy of them to the modeler’s servers. It is therefore preferable to do the training in a Federated Learning setting, wherein each user maintains a private database of her text messages securely on her own mobile device, and a shared global model is trained under the coordination of a central server based upon highly processed, minimally scoped, ephemeral updates from users [MMR+16, SS15]. These updates are high-dimensional vectors based on information from the user’s private database. Training a neural net is typically done by repeatedly iterating over these updates using a variant of a mini-batch stochastic gradient descent rule [CMBJ16, GBC16].

Although each update is ephemeral and contains no more (and typically significantly less) information than the user’s private database, a user might still be concerned about what information remains. In some circumstances, it is possible to learn individual words that a user has typed by inspecting that user’s most recent update. However, in the Federated Learning setting, the server does not need to access any individual user’s update in order to perform stochastic gradient descent; it requires only the element-wise weighted averages of the update vectors, taken over a random subset of users. Using a Secure Aggregation protocol to compute these weighted averages would ensure that the server may learn only that one or more users in this randomly selected subset wrote a given word, but not which users.

Federated Learning systems face several practical challenges. Mobile devices have only sporadic access to power and network connectivity, so the set of users participating in each update step is unpredictable and the system must be robust to users dropping out. Because the neural network may be parameterized by millions of values, updates may be large, representing a direct cost to users on metered network plans. Mobile devices also generally cannot establish direct communications channels with other mobile devices (relying on a server or service provider to mediate such communication) nor can they natively authenticate other mobile devices.

Thus, Federated Learning motivates a need for a Secure Aggregation protocol that:
1. operates on high-dimensional vectors
2. is highly communication efficient, even with a novel set of users on each instantiation
3. is robust to users dropping out
4. provides the strongest possible security under the constraints of a server-mediated, unauthenticated network model

7.3 Cryptographic Primitives

7.3.1 Secret Sharing

The protocol relies on Shamir’s $t$-out-of-$n$ Secret Sharing [Sha79], which allows a user to split a secret $s$ into $n$ shares, such that any $t$ shares can be used to reconstruct $s$, but any set of at most $t-1$ shares gives no information about $s$.

The scheme is parameterized over a finite field $\mathbb{F}$ of size at least $l > 2^k$ (where $k$ is the security parameter of the scheme), e.g. $\mathbb{F} = \mathbb{Z}_p$ for some large public prime $p$. Note that such a large field size is needed because the scheme requires clients to secret share their secret keys (whose length must be proportional to the security parameter for the security proof to go through). An additional assumption is that integers $1, \ldots, n$ (which will be used to denote users in the protocol) can be identified with distinct field elements in $\mathbb{F}$. Given these parameters, the scheme consists of two algorithms. The sharing algorithm $SS.share(s, t, \mathcal{U}) \rightarrow \{(u, s_u)\}_{u \in \mathcal{U}}$ takes as input a secret $s$, a set $\mathcal{U}$ of $n$ field elements representing user IDs, and a threshold $t \leq |\mathcal{U}|$; it produces a set of shares $s_u$, each of which is associated with a different $u \in \mathcal{U}$. The reconstruction algorithm $SS.recon(\{(u, s_u)\}_{u \in \mathcal{V}}, t) \rightarrow s$ takes as input the threshold $t$ and the shares corresponding to a subset $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| \geq t$, and outputs a field element $s$.

Correctness requires that $\forall s \in \mathbb{F}, \forall t, n$ with $1 \leq t \leq n, \forall \mathcal{U} \subseteq \mathbb{F}$ where $|\mathcal{U}| = n$, if $\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow SS.share(s, t, \mathcal{U}), \mathcal{V} \subseteq \mathcal{U}$ and $|\mathcal{V}| \geq t$, then $SS.recon(\{(u, s_u)\}_{u \in \mathcal{V}}, t) = s$. Security requires that $\forall s, s' \in \mathbb{F}$ and any $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| < t$:

$$\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow SS.share(s, t, \mathcal{U}) : \{(u, s_u)\}_{u \in \mathcal{V}} \equiv \\\\\\\\\\\\{\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow SS.share(s', t, \mathcal{U}) : \{(u, s_u)\}_{u \in \mathcal{V}}\}$$

where “$\equiv$” denotes that the two distributions are identical.

7.3.2 Key Agreement

Key Agreement consists of a tuple of algorithms

$(KA.param, KA.gen, KA.agree)$. The algorithm $KA.param(k) \rightarrow pp$ produces some public parameters (over which the scheme will be parameterized). $KA.gen(pp) \rightarrow (s_u^{SK}, s_u^{PK})$ allows any party $u$ to generate a private-public key pair. $KA.agree(s_u^{SK}, s_v^{PK}) \rightarrow s_{u,v}$ allows any user $u$ to combine their private key $s_u^{SK}$ with the public key $s_v^{PK}$ for any $v$ (generated using the same $pp$), to obtain a private shared key $s_{u,v}$ between $u$ and $v$.

The specific Key Agreement scheme used is Diffie-Hellman key agreement [DH76], composed with a hash function. More specifically, $KA.param(k) \rightarrow (G', g, q, H)$ samples group $G'$ of prime order $q$,
along with a generator \(g\), and a hash function \(H^2\); \textbf{KA.gen}(\(\mathbb{G}'\), \(g, q, H\)) \(\rightarrow\) \((x, g^x)\) samples a random \(x \leftarrow \mathbb{Z}_q\) as the secret key \(s_u^{SK}\), and \(g^x\) as the public key \(s_u^{PK}\); and \textbf{KA.agree}(\(x_u, g^{x_u}\)) \(\rightarrow\) \(s_{u,v}\) outputs \(s_{u,v} = H((g^{x_u})^{x_v})\).

Correctness requires that, for any key pairs generated by users \(u\) and \(v\) (using \textbf{KA.gen} and the same parameters \(pp\)), \textbf{KA.agree}(\(s_u^{SK}, s_v^{PK}\)) = \textbf{KA.agree}(\(s_v^{SK}, s_u^{PK}\)). For security, in the honest but curious model, it must be shown that for any adversary who is given two honestly generated public keys \(s_u^{PK}\) and \(s_v^{PK}\) (but neither of the corresponding secret keys \(s_u^{SK}\) or \(s_v^{SK}\)), the shared secret \(s_{u,v}\) computed from those keys is indistinguishable from a uniformly random string. This exactly mirrors the Decisional Diffie-Hellman (DDH) assumption from Definition 1.

In order to prove security against active adversaries (Theorem 10), a somewhat stronger security guarantee is needed for Key Agreement, namely that an adversary who is given two honestly generated public keys \(s_u^{PK}\) and \(s_v^{PK}\), and also the ability to learn \textbf{KA.agree}(\(s_u^{SK}, s_v^{PK}\)) and \textbf{KA.agree}(\(s_v^{SK}, s_u^{PK}\)) for any \(s^{PK}\)'s of its choice (but different from \(s_u^{PK}\) and \(s_v^{PK}\)), still cannot distinguish \(s_{u,v}\) from a random string. The following variant of the Oracle Diffie-Hellman assumption (ODH) [ABR01], referred to as the Two Oracle Diffie-Hellman assumption (2ODH), captures this stronger requirement:

**Definition 6** (Two Oracle Diffie-Hellman assumption (2ODH)). Let \(G(k) \rightarrow (\mathbb{G}', g, q, H)\) be an efficient algorithm which samples a group \(\mathbb{G}'\) of order \(q\) with generator \(g\), as well as a function \(H: \{0,1\}^* \rightarrow \{0,1\}^k\). Consider the following probabilistic experiment, parameterized by a PPT adversary \(M\), a bit \(b\) and a security parameter \(k\).

\(2\text{ODH-Exp}^b_{G,M}(k)\):

1. \((\mathbb{G}', g, q, H, A, B, s) \leftarrow G(k)
2. a \leftarrow \mathbb{Z}_q; A \leftarrow g^a
3. b \leftarrow \mathbb{Z}_q; B \leftarrow g^b
4. if \(b = 1\), \(s \leftarrow H(g^{ab})\), else \(s \leftarrow \mathbb{Z}_k\), \{0,1\}^k
5. \(M^{O_a(\cdot), O_b(\cdot)}(\mathbb{G}', g, q, H, A, B, s) \rightarrow b'\)
6. Output 1 if \(b = b'\), 0 o/w.

where \(O_a(X)\) returns \(H(X^a)\) on any \(X \neq B\) (and an error on input \(B\)) and similarly \(O_b(X)\) returns \(H(X^b)\) on any \(X \neq A\). The advantage of the adversary is defined as

\[
\text{Adv}_{G,M}^{2\text{ODH}}(k) := |\text{Pr}[2\text{ODH-Exp}^1_{G,M}(k) = 1] - \text{Pr}[2\text{ODH-Exp}^0_{G,M}(k) = 1]|\]

The Two Oracle Diffie-Hellman assumption holds for \(G\) if for all PPT adversaries \(M\), there exists a negligible function \(\epsilon\) such that \(\text{Adv}_{G,M}^{2\text{ODH}}(k) \leq \epsilon(k)\). This assumption can be directly used to prove the security property we need for Key Agreement: the two oracles \(O_a(\cdot), O_b(\cdot)\) formalize the ability of the adversary \(M\) to learn \textbf{KA.agree}(\(s_u^{SK}, s_v^{PK}\)) and \textbf{KA.agree}(\(s_v^{SK}, s_u^{PK}\)) for different \(s^{PK}\), and the negligible advantage of \(M\) in the above game corresponds to an inability to distinguish between \(s = s_{u,v} \leftarrow H(g^{ab})\), and \(s \leftarrow \mathbb{Z}_k, \{0,1\}^k\).

\(^2\text{In practice, one can use SHA-256.}\)
7.3.3 Authenticated Encryption

(Symmetric) Authenticated Encryption combines confidentiality and integrity guarantees for messages exchanged between two parties. It consists of a key generation algorithm that outputs a private key $d_{SK}$, an encryption algorithm $AE._{enc}$ that takes as input a key and a message and outputs a ciphertext, and a decryption algorithm $AE._{dec}$ that takes as input a ciphertext and a key and outputs either the original plaintext, or a special error symbol $\perp$. A correctness requirement is that for all keys $c \in \{0, 1\}^k$ and all messages $x$, $AE._{dec}(c, AE._{enc}(c, x)) = x$. For security, the requirement is indistinguishability under a chosen plaintext attack (IND-CPA) and ciphertext integrity (IND-CTXT) as defined in [BN00]. Informally, the guarantee is that for any adversary $M$ that is given encryptions of messages of its choice under a randomly sampled key $c$ (where $c$ is unknown to $M$), $M$ cannot distinguish between fresh encryptions under $c$ of two different messages, nor can $M$ create new valid ciphertexts (different from the ones it received) with respect to $c$ with better than negligible advantage.

7.3.4 Pseudorandom Generator

The protocol requires a secure Pseudorandom Generator [Yao82b, BM84] $PRG$ that takes in a uniformly random seed of some fixed length, and whose output space is $[0, R]^m$ (i.e. the input space for the protocol). Security for a Pseudorandom Generator guarantees that its output on a uniformly random seed is computationally indistinguishable from a uniformly sampled element of the output space, as long as the seed is hidden from the distinguisher.

7.3.5 Signature Scheme

The protocol relies on a standard UF-CMA secure signature scheme $(SIG._{gen}, SIG._{sign}, SIG._{ver})$. The key generation algorithm $SIG._{gen}(k) \rightarrow (d_{PK}, d_{SK})$ takes as input the security parameter and outputs a secret key $d_{SK}$ and a public key $d_{PK}$; the signing algorithm $SIG._{sign}(d_{SK}, m) \rightarrow \sigma$ takes as input the secret key and a message and outputs a signature $\sigma$; the verification algorithm $SIG._{ver}(d_{PK}, m, \sigma) \rightarrow \{0, 1\}$ takes as input a public key, a message and a signature, and returns a bit indicating whether the signature should be considered valid. For correctness, it is required that $\forall m,$

$$\Pr[(d_{PK}, d_{SK}) \leftarrow SIG._{gen}(k), \sigma \leftarrow SIG._{sign}(d_{SK}, m) : SIG._{ver}(d_{PK}, m, \sigma) = 1] = 1$$

Security demands that no PPT adversary, given a fresh honestly generated public key and access to an oracle producing signatures on arbitrary messages, should be able to produce a valid signature on a message on which the oracle was queried on with more than negligible probability.

7.3.6 Public Key Infrastructure

To prevent the server from simulating an arbitrary number of clients (in the active-adversary model), the security proof requires the support of a public key infrastructure that allows clients to register identities, and sign messages using their identity, such that other clients can verify this signature, but cannot

---

3Without loss of generality, it is assumed that the key generation algorithm samples keys as uniformly random strings.
impersonate them. In this model, each party $u$ will register $(u, d_u^{PK})$ to a public bulletin board during the setup phase. The bulletin board will only allow parties to register keys for themselves, so it will not be possible for the attacking parties to impersonate honest parties.

### 7.4 A Practical Secure Aggregation Protocol

The protocol is run (in a synchronous network) between a server and a set of $n$ users, and consists of four rounds. Each user $u$ holds as input a vector $x_u$ (of equal length $m$) consisting of elements from $\mathbb{Z}_R$ for some $R$. The server has no input, but can communicate with the users through secure (private and authenticated) channels. At any point, users can drop out of the protocol (in which case they stop sending messages completely), and the server will be able to produce a correct output as long as $t$ of them survive until the last round. To simplify the notation it is convenient to assume that each user $u$ is assigned a unique “logical identity” (also denoted with $u$) in the form of an integer between 1 and $n$, so that no two honest users share the same index.

A complete description is provided in Figure 7.1. Note that, in the figure, where it says that the server “collects messages from at least $t$ users”, it should be understood that the server receives the messages from all users that have not dropped out/aborted in that round (recall that the proof is in the synchronous setting), and aborts if the number of messages received is less than $t$. In a practical implementation, the server would wait until a specified timeout (considering all users who did not respond in time to have dropped out), and abort itself if not enough messages are received before such timeout.

To prove security in the active adversary model a PKI is invoked, which for simplicity is modelled by assuming all clients receive as input (from a trusted third party) public signing keys for all other clients.

Overall, the protocol is parameterized over a security parameter $k$, which can be adjusted to bound the success probability of any attacker. In all theorems, it is implicitly assumed that the number of clients $n$ is polynomially bounded in the security parameter. Moreover, some of the primitives also require additional global parameters.

Note that Figure 7.1 presents both variants of the protocol: in the honest but curious case, since all parties are following the protocol honestly, it is possible avoid the use of signatures and the need for a PKI (which, most notably, allows us to avoid the **ConsistencyCheck** round entirely).

### 7.5 Security Analysis

The following lemma will be used in the proofs below; it can be viewed as a restatement of the security of a one-time pad. The proof is omitted but can be done by induction on $n$.

---

4These identities will be bound to the users’ keys by a PKI. This is relied on in the active-adversary setting.
Secure Aggregation Protocol

- **Setup:**
  - All parties are given the security parameter $k$, the number of users $n$ and a threshold value $t$, honestly generated $pp \leftarrow \text{KA.gen}(k)$, parameters $m$ and $R$ such that $\mathbb{Z}_R$ is the space from which inputs are sampled, and a field $\mathbb{F}$ to be used for secret sharing. All users also have a private authenticated channel with the server.
  - All users $u$ receive their signing key $d_u^{SK}$ from the trusted third party, together with verification keys $d_u^{PK}$ bound to each user identity $u$.

- **Round 0 (AdvertiseKeys):**
  **User u:**
  - Generate key pairs $(c_u^{PK}, c_u^{SK}) \leftarrow \text{KA.gen}(pp), (s_u^{PK}, s_u^{SK}) \leftarrow \text{KA.gen}(pp)$, and generate $\sigma_u \leftarrow \text{SIG.sign}(d_u^{SK}, c_u^{PK} || s_u^{PK})$.
  - Send $(c_u^{PK} || s_u^{PK} || \sigma_u)$ to the server (through the private authenticated channel) and move to next round.

  **Server:**
  - Collect at least $n$ messages from individual users in the previous round (denote with $\mathcal{U}_1$ this set of users). Otherwise, abort.
  - Broadcast to all users in $\mathcal{U}_1$ the list $\{(v, c_v^{PK}, s_v^{PK}, \sigma_v)\}_{v \in \mathcal{U}_1}$ and move to next round.

- **Round 1 (ShareKeys):**
  **User u:**
  - Receive the list $\{(v, c_v^{PK}, s_v^{PK}, \sigma_v)\}_{v \in \mathcal{U}_1}$ broadcasted by the server. Assert that $|\mathcal{U}_1| \geq t$, that all the public key pairs are different, and that $\forall v \in \mathcal{U}_1, \text{SIG.} \text{ver}(d_v^{PK}, c_v^{PK} || s_v^{PK} || \sigma_v) = 1$.
  - Sample a random element $b_u \leftarrow \mathbb{F}$ (to be used as a seed for a PRG).
  - Generate $t$-out-of-$|\mathcal{U}_1|$ shares of $s_u^{SK}$: $\{(v, s_v^{SK})\}_{v \in \mathcal{U}_1} \leftarrow \text{SS.share}(s_u^{SK}, t, \mathcal{U}_1)$
  - Generate $t$-out-of-$|\mathcal{U}_1|$ shares of $b_u$: $\{(v, b_v)\}_{v \in \mathcal{U}_1} \leftarrow \text{SS.share}(b_u, t, \mathcal{U}_1)$
  - For each other user $v \in \mathcal{U}_1 \setminus \{u\}$, compute $e_{u,v} \leftarrow \text{AE.enc}(\text{KA.agree}(c_u^{SK}, c_v^{PK}), u || v || s_v^{SK} || b_{u,v})$
  - If any of the above operations (assertion, signature verification, key agreement, encryption) fails, abort.
  - Send all the ciphertexts $e_{u,v}$ to the server (each implicitly containing addressing information $u, v$ as metadata).
  - Store all messages received and values generated in this round, and move to the next round.

  **Server:**
  - Collect lists of ciphertexts from at least $t$ users (denote with $\mathcal{U}_2 \subseteq \mathcal{U}_1$ this set of users).
  - Sends to each user $u \in \mathcal{U}_2$ all ciphertexts encrypted for it: $\{e_{u,v}\}_{v \in \mathcal{U}_2}$ and move to the next round.

- **Round 2 (MaskedInputCollection):**
  **User u:**
  - Receive (and store) from the server the list of ciphertexts $\{e_{u,v}\}_{v \in \mathcal{U}_2}$ (and infer the set $\mathcal{U}_2$). If the list is of size $< t$, abort.
  - For each other user $v \in \mathcal{U}_2 \setminus \{u\}$, compute $s_{u,v} \leftarrow \text{KA.agree}(s_u^{SK}, s_v^{PK})$ and expand this value using a PRG into a random vector $p_{u,v} = \Delta_{u,v} \cdot \text{PRG}(s_{u,v})$, where $\Delta_{u,v} = 1$ when $u > v$, and $\Delta_{u,v} = -1$ when $u < v$ (note that $p_{u,v} + p_{v,u} = 0 \forall u \neq v$). Additionally, define $p_{u,u} = 0$.
  - Compute the user’s own private mask vector $p_u = \text{PRG}(b_u)$. Then, Compute the masked input vector $y_u \leftarrow x_u + p_u + \sum_{v \in \mathcal{U}_2} p_{u,v} \mod R$
  - If any of the above operations (key agreement, PRG) fails, abort. Otherwise, Send $y_u$ to the server and move to the next round.

  **Server:**
  - Collect $y_u$ from at least $t$ users (denote with $\mathcal{U}_3 \subseteq \mathcal{U}_2$ this set of users). Send to each user in $\mathcal{U}_3$ the list $\mathcal{U}_3$.
The honest-but-curious security argument considers executions of the secure aggregation protocol where the underlying cryptographic primitives are instantiated with security parameter $k$, a server $S$ interacts with a set $U$ of $n$ users (denoted with logical identities $1, \ldots, n$) and the threshold is set to $t$. In such executions, users might abort at any point during the execution, and we denote with $U_i$ the subset of the users that correctly sent their message to the server at round $i - 1$, such that $U \supseteq U_1 \supseteq U_2 \supseteq U_3 \supseteq U_4 \supseteq U_5$. For
example, users in $\mathcal{U}_2 \setminus \mathcal{U}_3$ are exactly those that abort before sending the message to the server in Round 2, but after sending the message of Round 1. If Round ConsistencyCheck has been omitted, define $\mathcal{U}_4 := \mathcal{U}_3$.

Denote the input of each user $u$ with $x_u$, and with $x_{\mathcal{U}'} = \{x_u\}_{u \in \mathcal{U}'}$ the inputs of any subset of users $\mathcal{U}' \subseteq \mathcal{U}$.

In such a protocol execution, the view of a party consists of its internal state (including its input and randomness) and all messages this party received from other parties (the messages sent by this party do not need to be part of the view because they can be determined using the other elements of its view). Moreover, if the party aborts, it stops receiving messages and the view is not extended past the last message received.

Given any subset $\mathcal{C} \subseteq \mathcal{U} \cup \{S\}$ of the parties, let $\text{REAL}^{t,t,k}_{\mathcal{C}}(x_{\mathcal{U}_1}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$ be a random variable representing the combined views of all parties in $\mathcal{C}$ in the above protocol execution, where the randomness is over the internal randomness of all parties, and the randomness in the setup phase.

**Theorem 8** (Honest But Curious Security, against clients only). There exists a PPT simulator $\text{SIM}$ such that for all $k, t, \mathcal{U}$ with $t \leq |\mathcal{U}|$, $x_{\mathcal{U}_1}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5$ and $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathcal{U}$, $\mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4 \supseteq \mathcal{U}_5$, the output of $\text{SIM}$ is perfectly indistinguishable from the output of $\text{REAL}^{t,t,k}_{\mathcal{C}}$:

$$\text{REAL}^{t,t,k}_{\mathcal{C}}(x_{\mathcal{U}_1}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5) \equiv \text{SIM}^{t,t,k}_{\mathcal{C}}(x_{\mathcal{C}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$$

**Proof.** Note that, since the view of the server is omitted, the joint view of the parties in $\mathcal{C}$ does not depend (in an information theoretic sense) on the inputs of the parties not in $\mathcal{C}$. The simulator can therefore produce a perfect simulation by running the honest but curious users on their true inputs, and all other users on a dummy input (for example, a vector of 0s), and outputting the simulated view of the users in $\mathcal{C}$. In more detail, the only value sent by the honest parties which depend on their input is $y_u$ (sent to the server in round MaskedInputCollection). One can easily note that the response sent by the server to the users in round MaskedInputCollection just contains a list of user identities which depends on which users respond on the previous round, but not on the specific $y_u$ values of the responses. This means that the simulator can use dummy values for the inputs of all honest parties not in $\mathcal{C}$, and the joint view of users in $\mathcal{C}$ will be identical to that in $\text{REAL}^{t,t,k}_{\mathcal{C}}$. \hfill \square

The following theorem security against an honest-but-curious server, who can additionally combine knowledge with some honest-but-curious clients. It will be shown that any such group of honest-but-curious parties can be simulated given the inputs of the clients in that group, and only the sum of the values of the remaining clients. Intuitively, this means that those clients and the server learn “nothing more” than their own inputs, and the sum of the inputs of the other clients. Additionally, if too many clients abort before Round Unmasking, then we show that we can simulate the view of the honest-but-curious parties given no information about the remaining clients’ values. Thus, in this case, the honest-but-curious parties learn nothing about the remaining clients’ values.

Importantly, the simulated view must contain fewer than $t$ honest-but-curious clients, or else we cannot guarantee security.

**Theorem 9** (Honest But Curious Security, with curious server). There exists a PPT simulator $\text{SIM}$ such that for all $t, \mathcal{U}, x_{\mathcal{U}_1}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4$, and $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathcal{U} \cup \{S\}$, $|\mathcal{C} \setminus \{S\}| < t$, $\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4$...
\( U_4 \subseteq U_5 \), the output of SIM is computationally indistinguishable from the output of REAL_{C,t,k}:

\[
\text{REAL}_{C,t,k}(x_U, U_1, U_2, U_3, U_4, U_5) \\
\approx_c \text{SIM}_{C,t,k}(x_C, z, U_1, U_2, U_3, U_4, U_5)
\]

where

\[
z = \begin{cases} 
\sum_{u \in U_3 \setminus C} x_u & \text{if } |U_3| \geq t \\
\perp & \text{otherwise.}
\end{cases}
\]

Proof. The proof proceeds by a standard hybrid argument, defining a simulator SIM through a series of (polynomially many) subsequent modifications to the random variable REAL, so that any two subsequent random variables are computationally indistinguishable.

Hyb₀ This random variable is distributed exactly as REAL, the joint view of the parties C in a real execution of the protocol.

Hyb₁ This hybrid changes the behavior of simulated honest parties in the set \( U_2 \setminus C \), so that instead of using \( \text{KA.agree}(c_u^{SK}, c_v^{PK}) \) to encrypt and decrypt messages to other users \( v \) in the same set, they use a uniformly random encryption key \( c_{u,v} \) chosen by the simulator. The Decisional Diffie-Hellman assumption (as recalled in Definition 1) guarantees that this hybrid is indistinguishable from the previous one.

Hyb₂ This hybrid substitutes all ciphertexts encrypted by honest parties in the set \( U_2 \setminus C \) and sent to other honest parties with encryptions of 0 (padded to the appropriate length) instead of shares of \( s_u^{SK} \) and \( b_u \). However, the honest clients in that set continue to respond with the correct shares of \( s_u^{SK} \) and \( b_u \) in Round Unmasking. Since only the contents of the ciphertexts have changed, IND-CPA security of the encryption scheme guarantees that this hybrid is indistinguishable from the previous one.

Hyb₃ Define:

\[
U^* = \begin{cases} 
U_2 \setminus C & \text{if } z = \perp \\
U_2 \setminus U_3 \setminus C & \text{otherwise.}
\end{cases}
\]

This hybrid is distributed exactly as the previous one, but substitutes all shares of \( b_u \) generated by parties \( u \in U^* \) and given to the corrupted parties in Round ShareKeys with shares of 0 (using a different sharing of 0 for every \( u \in U^* \)). Note that, in this hybrid and the previous one, the adversary does not receive any additional shares of \( b_u \) for users \( u \) in the set \( U^* \) in Round Unmasking, either because the honest clients do not reveal shares of \( b_u \) for such \( u \), or because all honest clients abort (when \( |U_3| < t \), which happens exactly when \( z = \perp \)). Thus, \( M_C \)'s joint view contains only \( |C| < t \) shares of each \( b_u \). The properties of Shamir’s secret sharing thus guarantee that the distribution of any \( |C| \) shares of 0 is identical to the distribution of an equivalent number of shares of any given secret \( b_u \), making this hybrid identically distributed to the previous one.

Hyb₄ In this hybrid, for all parties \( u \in U^* \), instead of computing \( p_u \leftarrow \text{PRG}(b_u) \), set it to be a uniformly random vector (of the appropriate size).
Note that, in the previous hybrid, since \( b_u \) is chosen uniformly at random and its shares given to the adversary are substituted with shares of 0, the output of the random variable does not depend on the seed of the PRG except through the PRG’s output. Therefore, the only change in this hybrid boils down to substituting the output of a PRG (on a randomly generated seed otherwise independent from the joint view of parties in \( C \)) with a uniformly random value. Therefore, from the security of the PRG, this hybrid is indistinguishable from the previous one.

**Hyb5** For all parties \( u \in \mathcal{U}^* \), in Round **MaskedInputCollection**, instead of sending:

\[
y_u \leftarrow x_u + p_u + \sum_{v \in \mathcal{U}_2} p_{u,v}
\]

send:

\[
y_u \leftarrow p_u + \sum_{v \in \mathcal{U}_2} P_{u,v}
\]

Since \( p_u \) was changed in the previous hybrid to be uniformly random and independent of any other values, \( x_u + p_u \) is also uniformly random, and so this hybrid and the previous hybrid are identically distributed. Further, this hybrid and all subsequent hybrids do not depend on the values \( x_u \) for \( u \in \mathcal{U}^* \).

**Note:** If \( z = \perp \), then we can ignore the further hybrids, and let \( \text{SIM} \) be as described in **Hyb5**, since \( \text{SIM} \) can already simulate \( \text{REAL} \) without knowing \( x_u \) for any \( u \not\in C \). Therefore in the following hybrids it is assumed \( z \neq \perp \).

**Hyb6** This hybrid substitutes all shares of \( s^{SK}_u \) generated by parties \( u \in \mathcal{U}_3 \setminus C \) and given to the corrupted parties in Round **ShareKeys** with shares of 0 (using a different sharing of 0 for every \( u \in \mathcal{U}_3 \setminus C \)). Following an analogous argument to that for **Hyb3**, the properties of Shamir’s secret sharing guarantee that this hybrid is identically distributed to the previous one.

**Hyb7** Fix a specific user \( u' \in \mathcal{U}_3 \setminus C \). For this user, and each other user \( u \in \mathcal{U}_3 \setminus C \), in order to compute the value \( y_u \) sent to the server, substitute the joint noise key (which would be computed by \( u' \) and \( u \) as \( s_{u',u} = s_{u,u'} \leftarrow \text{KA.agree}(s^{SK}_{u'}, s^{PK}_u) \)) with a uniformly random value (which will be used by both parties as a PRG seed).

In more detail, for each user \( u \in \mathcal{U}_3 \setminus C \setminus \{u'\} \), a value \( s'_{u',u} \) is sampled uniformly at random and, instead of sending

\[
y_u \leftarrow x_u + p_u + \sum_{v \in \mathcal{U}_2} p_{u,v}
\]

\( \text{SIM} \) sends

\[
y'_u \leftarrow x_u + p_u + \sum_{v \in \mathcal{U}_2 \setminus \{u'\}} p_{u,v} + \Delta_{u,u'} \cdot \text{PRG}(s'_{u',u})
\]

and accordingly

\[
y'_{u'} \leftarrow x_{u'} + p_{u'} + \sum_{v \in \mathcal{U}_2} \Delta_{u',v} \cdot \text{PRG}(s'_{u',v})
\]

where \( \Delta_{u,v} = 1 \) when \( u > v \) and \( \Delta_{u,v} = -1 \) when \( u < v \).

It is easy to see that the Decisional Diffie-Hellman Assumption (Definition 1) guarantees that this hybrid is indistinguishable from the previous one.
Hyb₈ In this hybrid, for the same party \( u' \) chosen in the previous hybrid and all other parties \( v \in \mathcal{U}_3 \setminus C \), instead of computing \( p_{u',v} \leftarrow \Delta_{u',v} \cdot \text{PRG}(s_{u',v}) \), compute it using fresh randomness \( r_{u',v} \) (of the appropriate size) as \( p_{u',v} \leftarrow \Delta_{u',v} \cdot r_{u',v} \).

Note that, in the previous hybrid, since \( s_{u',v} \) is chosen uniformly at random (and independently from the Diffie-Hellman keys), the output of the random variable does not depend on the seed of the PRG except through the PRG’s output. Therefore, the only change in this hybrid boils down to substituting the output of a PRG (on an randomly generated seed otherwise independent from the joint view of parties in \( C \)) with a uniformly random value. Therefore, from the security of the PRG, this hybrid is indistinguishable from the previous one.

Hyb₉ In this hybrid, for all users \( u \in \mathcal{U}_3 \setminus C \), in round MaskedInputCollection instead of sending:

\[
y_u \leftarrow x_u + p_u + \sum_{v \in \mathcal{U}_2} p_{u,v} = x_u + p_u + \sum_{v \in \mathcal{U}_3 \setminus C} p_{u,v} + \sum_{v \in \mathcal{U}_2 \setminus \{u\} \setminus C} p_{u,v}
\]

send:

\[
y_u \leftarrow w_u + p_u + \sum_{v \in \{u\} \setminus \{u\} \setminus C} p_{u,v}
\]

Where \( \{w_u\}_{u \in \mathcal{U}_3 \setminus C} \) are uniformly random, subject to \( \sum_{u \in \mathcal{U}_3 \setminus C} w_u = \sum_{d \in \mathcal{U}_3 \setminus C} x_u = z \). Invoking Lemma 6 with \( n = |\mathcal{U}_3 \setminus C| \), this hybrid is identically distributed to the previous one. Moreover, note that to sample from the random variable described by this hybrid, knowledge of the individual \( x_u \) for \( u \in \mathcal{U}_3 \setminus C \) is not needed, and their sum \( z \) is sufficient.

Now define a PPT simulator \( \text{SIM} \) that samples from the distribution described in the last hybrid. The argument above proves that the output of the simulator is computationally indistinguishable from the output of REAL, completing the proof.

\[\square\]

7.5.2 Privacy against Active Adversaries

This section presents the argument showing privacy against active adversaries. Active adversaries in this context are parties (clients or the server) that deviate from the protocol, sending incorrect and/or arbitrarily chosen messages to honest users, aborting, omitting messages, and sharing their entire view of the protocol with each other, and also with the server (if the server is also an active adversary).

The following security argument shows only input privacy for honest users: it is much harder to additionally guarantee correctness and availability for the protocol when some users are actively adversarial. Such users can maul the output of the protocol by setting their input values \( x_u \) to be out of range\(^5\), by sending inconsistent Shamir shares to other users in Round ShareKeys, or by reporting incorrect shares to the server in Round Unmasking. Making such deviations efficient to detect and possibly recover from is left to future work.

There are a few key differences between the argument for honest-but-curious security, and the argument for privacy against active adversaries.

\(^5\)Typically, each element of \( x_u \) is expected to be from a range \( [0, R_U) \subset [0, R) \), such that the sum of all \( x_u \) is in \( [0, R) \). However, an actively adversarial user could choose \( x_u \) outside the expected range, i.e. on \( [R_U, R) \), allowing the adversarial user disproportionate impact on protocol’s result, thus undermining correctness.
The first key difference is that the proof against active adversaries assumes that there exists a public-key infrastructure (PKI), which guarantees to users that messages they receive came from other users (and not the server). Without this assumption, the server can perform a Sybil attack on the users in Round ShareKeys, by simulating for a specific user \( u \) all other users \( v \) in the protocol and thus receiving all \( u \)’s key shares and recovering that users’ input.

By assuming a PKI the server’s power in the remainder of the protocol is reduced to lying to users about which other users have dropped out: since all user-to-user messages (sent in round ShareKeys) are authenticated through an authenticated encryption scheme, the server cannot add, modify or substitute messages, but rather, can only fail to deliver them. Note, importantly, that the server can try to give a different view to each user of which other users have dropped out of the protocol. In the worst case, this could allow the server to learn a different set of shares from each user in Round Unmasking, allowing it to potentially reconstruct more secrets than it should be allowed to. The ConsistencyCheck round is included in the protocol to deal with this issue. The inclusion of the ConsistencyCheck round is the second key difference with the honest-but-curious proof.

The other key difference is that the proof will be in the random oracle (RO) model. To see why this necessary, notice that honestly acting users are essentially “committed” to their secrets and input by the end of the MaskedInputCollection round. However, the server can adaptively choose which users drop after the MaskedInputCollection round. This causes problems for a simulation proof, because the simulator doesn’t know honest users’ real inputs, and must use dummy information in the earlier rounds, thus “committing” itself to wrong values that are potentially easily detectable. The random oracle adds a trapdoor for the simulator to equivocate, so that even if it commits to dummy values in early rounds, it can reprogram the random oracle to make the dummy values indistinguishable from honest users’ values. More details can be seen in the proof of Theorem 10.

In order to leverage the security of the key agreement in a context where some of the clients might be active adversaries, the following theorem relies on a slight variant of the Oracle Diffie-Hellman assumption (ODH) [ABR01], which is referred to as the Two Oracle Diffie-Hellman assumption (2ODH).

**Theorem 10** (Privacy against active adversaries, including the server). There exists a PPT simulator \( \text{SIM} \) such that for all \( k, t, \mathcal{U}, \mathcal{C} \subseteq \mathcal{U} \cup \{S\} \) and \( x_{\mathcal{U}\setminus\mathcal{C}} \), letting \( n = |\mathcal{U}| \) and \( n_C = |\mathcal{C} \cap \mathcal{U}| \), if \( 2t > n + n_C \), then the output of \( \text{SIM} \) is computationally indistinguishable from the output of \( \text{REAL}^{\mathcal{U},t,k} \):

\[
\text{REAL}^{\mathcal{U},t,k}(M_C, x_{\mathcal{U}\setminus\mathcal{C}}) \approx_c \text{SIM}_c^{\mathcal{U},t,k}(x_{\mathcal{U}\setminus\mathcal{C}}) (M_C)
\]

where \( \delta = t - n_C \).

**Proof.** The proof proceeds by a standard hybrid argument, defining a simulator \( \text{SIM} \) through a series of (polynomially many) subsequent modifications to the real execution \( \text{REAL} \), so that the views of \( M_C \) in any two subsequent executions are computationally indistinguishable. In each of the hybrids below, even though it is not explicitly stated, \( \text{SIM} \) will cause honest parties to abort as they would during the real the protocol (e.g., if they receive a malformed message), and also if they are in a set \( \mathcal{U}_i \) output by \( M_C \).

**Hyb_0** This random variable is distributed exactly as the view of \( M_C \) in \( \text{REAL} \), the joint view of the parties \( \mathcal{C} \) in a real execution of the protocol.

**Hyb_1** In this hybrid, the real execution is emulated by a simulator that knows all the inputs \( x_u \) of the honest parties, and runs a full execution of the protocol with \( M_C \), which includes simulating the
random oracle “on the fly” (using a dynamically generated table), the PKI and the rest of the setup phase.

The view of the adversary in this hybrid is the same as the previous one.

Hyb$_2$ In this hybrid, the simulator additionally aborts if $M_C$ provides any of the honest parties $u$ (in round AdvertiseKeys) with a correct signature with respect to an honest $v$’s public key, on $(c^u_P||s^v_P)$ different from those sent by $v$. Since this amounts to breaking the security of the signature scheme, this hybrid is identical from the previous one.

Hyb$_3$ This hybrid is identical to Hyb$_2$, except that, for any pair of honest users $u, v$, the messages among them are encrypted (in round ShareKeys, before being given to $M_C$) and decrypted (in round Unmasking, after $M_C$ has delivered them) using a uniformly random key (as opposed to the one obtained through the key agreement $KA.agree(c^u_{SK}, c^v_P)$).

The 2ODH assumption guarantees that this hybrid is indistinguishable from the previous one. In particular, it is possible to switch the encryption keys between one pair of honest users at a time (since $n$ is polynomial in $k$, there are only polynomially many pairs of honest users), and argue that an adversary noticing the difference when one key is switched will also be able to break the 2ODH.

Hyb$_4$ This hybrid is identical to Hyb$_3$, except additionally, SIM will abort if $M_C$ succeeds to deliver, in round ShareKeys, a message to an honest client $u$ on behalf of another honest client $v$, such that i) the message is different from the message SIM had given $M_C$ in round ShareKeys, and ii) the message does not cause the decryption algorithm (using the proper key) to fail. Note that, as the encryption key that the two users were using in the previous hybrid was randomly selected, such a message would directly constitute a forgery against the INT-CTX security of the encryption scheme.

Hyb$_5$ In this hybrid, in addition, SIM substitutes all the encrypted shares sent between pairs of honest users with encryptions of 0. (It still returns the “real” shares in Round Unmasking as it did before).

Note that, since the corresponding encryption keys were chosen uniformly at random, IND-CPA security of the encryption scheme guarantees this hybrid is indistinguishable from the previous one.

Hyb$_6$ In this hybrid, in addition, SIM aborts if $M_C$ provides any of the honest parties (in round ConsistencyCheck) with a signature on a set which correctly verifies w.r.t. the public key of an honest party, but such that the honest client never produced a signature on that set.

Because of the security of the signature scheme, such forgeries can happen only with negligible probability, therefore this hybrid is indistinguishable from the previous one.

Now define the set $Q$ to be the only set $Q \subseteq \mathcal{U}$ such that there exists an honest user which received the set $Q$ in round ConsistencyCheck, and later received at least $t$ valid signatures on it in round Unmasking (where valid means that the signatures verify with respect to a set of distinct public signature keys among those received by the client from the trusted party at the start of the protocol).

In case no such set $Q$ exists (e.g. no set had enough signatures, or not enough honest users survived), define $Q = \emptyset$. 

118
Hyb\(_7\) In this hybrid, in addition, SIM aborts if \(M_C\) queries the random oracle/PRG on input \(b_u\) for some honest user \(u\) (i.e. the value sampled by SIM on behalf of \(u\) in round \(\text{ShareKeys}\)) either i) before the adversary received the responses from the honest players in round \(\text{Unmasking}\) or ii) after such responses have been received, but where \(u \notin Q\).

In both cases, because the value \(b_u\) is information theoretically hidden from \(M_C\), SIM will abort due to this new condition only if \(M_C\) is able to guess one of the \(b_u\), which can only happen with negligible probability (as they are chosen from the exponentially large domain \(\mathbb{F}\)). To see why the view of \(M_C\) does not depend on \(b_u\), let us analyze which of the view’s components depend on any \(b_u\). In case i), \(M_C\) only receives from SIM at most \(n_C\) shares of \(b\) (sent by \(u\) in round \(\text{ShareKeys}\), one for each of the corrupt clients). However, since \(n_C < t\), the distribution of any such shares is independent from \(b_u\) (because of the properties of secret sharing). Even in case ii), the view of \(M_C\) is still independent from \(b_u\); since \(u \notin Q\), no honest user would send to the server any share of \(b_u\), and therefore SIM does not have to send any to \(M_C\).

Hyb\(_8\) In this hybrid, in addition, SIM aborts if \(M_C\) queries the random oracle/PRG on input \(s_{u,v}\) for some honest users \(u, v\) either i) before the adversary received the responses from the honest players in round \(\text{Unmasking}\) or ii) after such responses have been received, but where \(u, v \in Q\).

By reduction to the 2ODH assumption this hybrid is indistinguishable from the previous one. In particular, consider a distinguisher \(\text{SIM}'\) which receives a 2ODH challenge \((G', g, q, A, B, z)\) and guesses at random two honest users \(u, v\), hoping that the adversary’s query which will cause the simulator to abort will be exactly \(s_{u,v}\). \(\text{SIM}'\) acts exactly as SIM in the previous hybrid, except it sets up \(s^{PK}_u = A\) and \(s^{PK}_v = B\) as the public keys for those users and uses its two oracles to complete the simulation without having access to the corresponding secret keys. In particular, in round \(\text{AdvertiseKeys}\), SIM' sends these modified public keys to \(M_C\) (as opposed to the fresh ones SIM would have sampled in the previous hybrid). In round \(\text{ShareKeys}\), rather than generating shares of the secret keys \(s^{SK}_u\) and \(s^{SK}_v\) (which it does not know), it generates and sends to the corrupt parties shares of 0. In round \(\text{MaskedInputCollection}\), when generating \(y\) values for all the honest users (to be sent to \(M_C\)), SIM' sets \(s_{u,v} = z\), and uses its two oracles \(O_u\) and \(O_v\) to compute all other required \(s\) values for \(u\) and \(v\) and other users. Then, if \(M_C\) makes a random oracle query for \(z\), SIM' will guess that \(z = H(g^{ab})\) and abort the simulation; otherwise it will guess that \(z\) was chosen at random.

Let us now analyze the advantage of such \(\text{SIM}'\) in the 2ODH-Exp game. Notice that, conditioned on the choice of \(u, v\) being correct, and until the point where the adversary makes a random oracle query for \(z\), the view of the adversary in this simulated protocol execution is exactly the same as the one of Hyb\(_7\). This is because, as in the previous argument, for both possible values of \(z\), the adversary will obtain less than \(t\) shares of both \(s^{SK}_u\) and \(s^{SK}_v\), which thus reveal “no information” about the actual values of \(s^{SK}_u\) and \(s^{SK}_v\). Moreover, by modeling the PRG as a random oracle, \(M_C\) cannot extract any information about \(s_{u,v}\) from \(y_u\) and \(y_v\) without querying the random oracle.
Therefore, if $M_C$ can distinguish between $\text{Hyb}_7$ and $\text{Hyb}_8$ with more than negligible probability, then it must be triggering the abort condition with more than negligible probability and therefore (conditioned on the choice of $u$ and $v$ being correct) $M_C$ must make to the random oracle/PRG a query of the form $H(g^{ab})$ with more than negligible probability. This implies that, when $z = H(g^{ab})$, $\text{SIM}'$ will claim (correctly) that $z = H(g^{ab})$ with non negligible probability. On the other hand, when $z$ is chosen uniformly random, it is information theoretically hidden from $M_C$’s view, and therefore $M_C$ can only make a query for it (which will cause $\text{SIM}'$ to incorrectly claim that $z = H(g^{ab})$) with negligible probability. In other words, if $M_C$ distinguishes between $\text{Hyb}_7$ and $\text{Hyb}_8$ with non-negligible probability $p$, then the algorithm $\text{SIM}'$ described above also breaks 2ODH assumption probability at least $p/2n^2$, which is non-negligible, concluding the argument.

$\text{Hyb}_9$ This hybrid is defined exactly as the previous one, except that the values of $y_u$ computed by the simulator on behalf of the honest clients and sent to $M_C$ in round MaskedInputCollection are substituted with uniformly sampled values, and the output of some random oracle queries for the PRG is modified to ensure consistency/correctness for the result. More in detail, after the server delivers to honest clients the messages for round ConsistencyCheck, but before SIM sends their responses, these messages sent by $M_C$ to the honest clients define a set $Q$ (as defined in hybrid $\text{Hyb}_6$). For all $u \in Q \setminus C$, SIM programs the random oracle to set $\text{PRG}(b_u)$ as follows:

$$\text{PRG}(b_u) \leftarrow y_u - x_u - \sum_{v \in F_u} \text{PRG}(s_{u,v})$$

where $v \in F_u$ iff $v \notin Q \setminus C$ and $M_C$ delivered a ciphertext to $u$ from $v$ in round ShareKeys (which captures the fact that in a real execution $u$ would have included the joint noise $p_{u,v}$ for $v$ in its masked input vector $y_u$). For all $u \notin Q \setminus C$, SIM sets $\text{PRG}(b_u)$ arbitrarily.

The view of $M_C$ in this hybrid is statistically indistinguishable from the previous one. First, note that for honest clients $u \notin Q$, since $M_C$ cannot query the PRG on input $b_u$, in both hybrids the value $y_u$ is distributed uniformly at random (and independent from the rest of the view).

Similarly, for honest clients $u \in Q$, before Round Unmasking, $M_C$ cannot query the PRG on input $b_u$, so $y_u$ looks uniformly random as expected. After Round Unmasking, when $M_C$ learns $b_u$, it has exactly the same distribution as in the previous hybrid, i.e. it satisfies

$$y_u = \text{PRG}(b_u) - \sum_{v \in F_u} \text{PRG}(s_{u,v}) = x_u$$

Thus, this hybrid is indistinguishable from the previous one.

$\text{Hyb}_{10}$ This hybrid is defined exactly as the previous one, except that for all $u \in Q \setminus C$, instead of programming the random oracle to set $\text{PRG}(b_u)$ to $\text{6}$:

$$\text{PRG}(b_u) \leftarrow$$

$$\begin{align*}
\text{PRG}(b_u) & \leftarrow y_u - \sum_{v \in F_u} \text{PRG}(s_{u,v}) \\
& = y_u - x_u - \sum_{v \in Q} \text{PRG}(s_{u,v}) \\
& - \sum_{v \in F_u \setminus Q} \text{PRG}(s_{u,v})
\end{align*}$$

\(\text{6}\)Notice that $Q$ must be a subset of $F_u$, or else $u$ aborts
as in the previous hybrid, SIM instead sets

\[ \text{PRG}(b_u) \leftarrow y_u - w_u - \sum_{v \in F_u \setminus Q} \text{PRG}(s_{u,v}) \]

where \( \{w_u\}_{u \in Q \setminus C} \) are chosen uniformly at random, subject to \( \sum_{u \in Q \setminus C} w_u = \sum_{u \in Q \setminus C} x_u \). Since, as argued before, \( s_{u,v}'s \) for \( u, v \in Q \setminus C \) are never queried by \( M_C \), by Lemma 6, in the view of \( M_C \), the above values are identically distributed as the previous hybrid.

Hyb\(_{11}\) This hybrid is defined as the previous one, with the only difference being that the simulator now does not receive the inputs of the honest parties, but instead, in round Unmasking, makes a query to the functionality Ideal for the set \( Q \setminus C \) and uses the value to sample the required \( w_u \) values. Note that since by construction \( |Q| \geq t \), \( |Q \setminus C| \geq t - n_C = s \), and therefore the functionality Ideal will not return \( \bot \).

It is easy to see that this change does not modify the view seen by the adversary, and therefore it is perfectly indistinguishable from the previous one. Moreover, this hybrid does not make use of the honest party’s inputs, and this concludes the proof.

\[ \square \]

### 7.5.3 Interpretation of Results

<table>
<thead>
<tr>
<th>Threat model</th>
<th>Minimum threshold</th>
<th>Minimum inputs in sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-only adversary</td>
<td>( \lceil \frac{n}{2} \rceil + 1 )</td>
<td>( t )</td>
</tr>
<tr>
<td>Server-only adversary</td>
<td>( \lceil \frac{n}{2} \rceil + 1 )</td>
<td>( t )</td>
</tr>
<tr>
<td>Clients-Server collusion</td>
<td>( \lceil \frac{n}{2} \rceil + 1 )</td>
<td>( t - n_C )</td>
</tr>
</tbody>
</table>

Figure 7.2: Parameterization for different threat models. “Minimum threshold” denotes the minimum value of \( t \) required for security in the given threat model. “Minimum inputs in the sum” denotes a lower bound on the number of users’ values that are included in the sum learned by the server. \( n \) denotes the total number of users, while \( n_C \) is the number of corrupt users.

**Security against only clients**

In each of the above theorems, observe that the joint view of any subset of clients, honest or adversarial, can be simulated given no information about the values of the remaining clients. This means, no matter how the \( t \) parameter is set, clients on their own learn nothing about other clients.

**Security against only the server**

From Theorems 9 and 10, observe that by setting \( n_C = 0 \), that is, there are no clients who cheat or collaborate with the server, setting \( t \geq \lceil \frac{n}{2} \rceil + 1 \) guarantees that the sum learned by the server contains the values of at least \( t > \frac{n}{2} \) clients, and the protocol can deal with up to \( \lceil \frac{n}{2} \rceil - 1 \) dropouts.
Security against a server colluding with clients

From Theorems 9 and 10, the server can be allowed to collude with up to $n_C = \lceil \frac{n}{3} \rceil - 1$ users, by setting $t \geq \lceil \frac{2n}{3} \rceil + 1$, at the same time guaranteeing that the sum learned by the server contains the values of at least $\frac{n}{3}$ clients. Additionally, the protocol is robust to up to $\lceil \frac{n}{3} \rceil - 1$ users dropping out.

7.6 Evaluation

Table 7.3 summarizes the performance of the secure aggregation protocol. All calculations below assume a single server and $n$ users, where each user holds a data vector of size $m$. The results reported below concern only the honest-but-curious version of the protocol, and ignore the cost of the PKI, all signatures, and Round ConsistencyCheck. Including these costs would not change any of the asymptotics, and would only slightly increases the computation and communication costs.

7.6.1 Performance Analysis of Client

**Computation cost:** $O(n^2 + mn)$. Each user $u$’s computation cost can be broken up as (1) Performing the $2n$ key agreements, which take $O(n)$ time, (2) Creating $t$-out-of-$n$ Shamir secret shares of $s_u^{SK}$ and $b_u$, which is $O(n^2)$ and (3) Generating values $p_u$ and $p_{u,v}$ for every other user $v$ for each entry in the input vector by stretching one PRG seed each, which takes $O(mn)$ time in total. Overall, each user’s computation is $O(n^2 + mn)$.

**Communication cost:** $O(n + m)$. The communication costs of each user can be broken up into 4 parts: (1) Exchanging keys with each other user by sending $2$ and receiving $2(n-1)$ public keys, (2) Sending $2(n-1)$ and receiving $2(n-1)$ encrypted secret shares, (3) Sending a masked data vector of size $m[\log_2 R]$ to the server, and (4) Sending the server $n$ secret shares, for an overall communication cost of $2na_K + (5n - 4)a_S + m[\log_2 R]$, where $a_K$ and $a_S$ are the number of bits in a key exchange public key and the number of bits in a secret share, respectively. Overall, the user’s communication complexity is $O(n + m)$. Assuming inputs for each user are on the same range $[0, R_U - 1]$, $R = n(R_U - 1) + 1$ is needed to avoid overflow. A user could transmit its raw data using $m[\log_2 R_U]$ bits. Taking $a_K = a_S = 256$ bits implies a communication expansion factor of $\frac{256(7n-4)+m[\log_2 R]}{m[\log_2 R_U]}$. For $R_U = 2^{16}$ (i.e. 16-bit input values), $m = 2^{20}$ elements, and $n = 2^{10}$ users, the expansion factor is $1.73 \times$; for $n = 2^{14}$ users, it is $3.62 \times$. For $m = 2^{24}$ elements and $n = 2^{14}$ users, the expansion factor is $1.98 \times$.

**Storage cost:** $O(n + m)$. The user must store the keys and secret-shares sent by each other user, which are $O(n)$ in total, and the data vector (which it can mask in-place), which has size $O(m)$.
Figure 7.4: Client Running Time and Data Transfer Costs. All wall-clock running times are for a single-threaded client implemented in Java, and ignore communication latency. Plotted points represent averages over 10 end-to-end iterations, and error bars represent 95% confidence intervals. (Error bars are omitted where measured standard deviation was less than 1%).

7.6.2 Performance Analysis of Server

Computation cost: $O(mn^2)$. The server’s computation cost can be broken down as (1) Reconstructing $n$ $t$-out-of-$n$ Shamir secrets (one for each user), which takes total time $O(n^2)$, and (2) generating and removing the appropriate $p_{u,v}$ and $p_u$ values from the sum of the $y_u$ values received, which takes time $O(mn^2)$ in the worst case.

Unfortunately, reconstructing $n$ secrets in the Shamir scheme takes $O(n^3)$ time in the general case: each secret reconstruction $SS.recon(\{(u, s_u)\}_{u \in U'}, t) \rightarrow s$ amounts to interpolating a polynomial $L$ over the points encoded by the shares and then evaluating at 0, which can be accomplished via Lagrange polynomials:

$$s = L(0) = \sum_{u \in U'} s_u \prod_{v \in U' \setminus \{u\}} \frac{v}{v - u} \pmod{p}$$

Each reconstruction requires $O(n^2)$ computation and must perform $n$ reconstructions, implying $O(n^3)$ total time. However, in this setting, it is possible to perform all of the reconstructions in $O(n^2)$ time by observing that all of the secrets will be reconstructed from identically-indexed sets of secret shares – that

1The server can reconstruct $n$ secrets from aligned $(t, n)$-Shamir shares in $O(t^2 + nt)$ by caching Lagrange coefficients; see section 7.6.2 for details.
(a) Wall-clock running time for the server, as the number of clients increases. The data vector size is fixed to 100K entries.

(b) Wall-clock running time for the server, as the size of the data vector increases. The number of clients is fixed to 500.

Figure 7.5: Server Running Time and Data Transfer Costs. All wall-clock running times are for a single-threaded server implemented in Java, and ignore communication latency. Plotted points represent averages over 10 end-to-end iterations. Error bars are omitted where measured standard deviations are less than 1%.

is, $\mathcal{U}$ is fixed across all secrets, because in round Unmasking, each user that is still alive sends a share of every secret that needs to be reconstructed. Lagrange basis polynomials

$$\ell_u = \prod_{v \in \mathcal{U} \setminus \{u\}} \frac{v}{v - u} \pmod{p}$$

can be precomputed using $O(n^2)$ time and $O(n)$ space. Using this, each of $n$ secrets can be reconstructed in $O(n)$ time as $L(0) = \sum_{u \in \mathcal{U}} s_u \ell_u \pmod{p}$ resulting in a total computational cost of $O(n^2)$ to reconstruct all the secrets.

The $O(mn^2)$ term can be broken into $O(m(n - d) + md(n - d))$, where $d$ is the number of users that dropped from the protocol. In practice, $d$ may be significantly smaller than $n$, which would also reduce the server’s computation cost.

**Communication cost:** $O(n^2 + mn)$. The server’s communication cost is dominated by its mediation of all pairwise communications between users, which is $O(n^2)$, and also for receiving masked data vectors from each user, which is $O(mn)$ in total.

**Storage cost:** $O(n^2 + m)$. The server must store $t$ shares for each user, which is $O(n^2)$ in total, along with an $m$-element buffer in which to maintain a running sum of $y_u$ as they arrive.

### 7.6.3 Prototype Performance

In order to measure performance a prototype was implemented in Java, using the following cryptographic primitives:

- For Key Agreement, Elliptic-Curve Diffie-Hellman over the NIST P-256 curve, composed with a SHA-256 hash.
- For Authenticated Encryption, AES-GCM with 128-bit keys.
- For the Pseudorandom Number Generator, AES in counter mode.
<table>
<thead>
<tr>
<th>Num. Clients</th>
<th>Dropouts</th>
<th>AdvertiseKeys</th>
<th>ShareKeys</th>
<th>MaskedInputColl.</th>
<th>Unmasking</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>500</td>
<td>0%</td>
<td>1 ms</td>
<td>154 ms</td>
<td>694 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>Server</td>
<td>500</td>
<td>0%</td>
<td>1 ms</td>
<td>26 ms</td>
<td>723 ms</td>
<td>1268 ms</td>
</tr>
<tr>
<td>Server</td>
<td>500</td>
<td>10%</td>
<td>1 ms</td>
<td>29 ms</td>
<td>623 ms</td>
<td>61586 ms</td>
</tr>
<tr>
<td>Server</td>
<td>500</td>
<td>30%</td>
<td>1 ms</td>
<td>28 ms</td>
<td>514 ms</td>
<td>142847 ms</td>
</tr>
<tr>
<td>Client</td>
<td>1000</td>
<td>0%</td>
<td>1 ms</td>
<td>336 ms</td>
<td>1357 ms</td>
<td>5 ms</td>
</tr>
<tr>
<td>Server</td>
<td>1000</td>
<td>0%</td>
<td>6 ms</td>
<td>148 ms</td>
<td>1481 ms</td>
<td>3253 ms</td>
</tr>
<tr>
<td>Server</td>
<td>1000</td>
<td>10%</td>
<td>6 ms</td>
<td>143 ms</td>
<td>1406 ms</td>
<td>179320 ms</td>
</tr>
<tr>
<td>Server</td>
<td>1000</td>
<td>30%</td>
<td>8 ms</td>
<td>143 ms</td>
<td>1169 ms</td>
<td>412446 ms</td>
</tr>
</tbody>
</table>

Figure 7.6: CPU wall clock times per round. All wall-clock running times are for a single-threaded servers and clients implemented in Java, and ignore communication latency. Each entry represents the average over 10 iterations. The data vector size is fixed to 100K entries with 24 bit entries.

<table>
<thead>
<tr>
<th>Num. Clients</th>
<th>Total Runtime Per-Client</th>
<th>StdDev</th>
<th>Server Total Runtime</th>
<th>StdDev</th>
<th>Total Communication P</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>13159 ms</td>
<td>6443 ms</td>
<td>14670 ms</td>
<td>6574 ms</td>
<td>0.95 MB</td>
</tr>
<tr>
<td>1000</td>
<td>23497 ms</td>
<td>6271 ms</td>
<td>27855 ms</td>
<td>6874 ms</td>
<td>1.15 MB</td>
</tr>
</tbody>
</table>

Figure 7.7: End-to-End running time for the protocol, executed over a wide-area-network. All running times are for a single-threaded servers and clients running in geographically separated datacenters, and include computation time, network latency, and time spent waiting for other participants. Each entry represents the average over 15 iterations, with iterations more than 3 standard deviations from the mean discarded. The data vector size is fixed to 100K entries with 62 bits per entry, and there are no induced dropouts (beyond <1% that occurred “naturally”).

An honest-but-curious setting is assumed, and thus the portions of Figure 7.1 special to active clients are omitted from the simulations. These omissions would not change the overall shape of the results in practice since the bulk of the costs involve masking, storing and sending the large data vector. This is discussed in more detail below.

An additional assumption is that when clients drop out of the protocol they drop after sending their shares to all other clients, but before sending their masked input to the server. This is essentially the “worst case” dropout, since all other clients have already incorporated the dropped clients’ masks, and the server must perform an expensive recovery computation to remove them. It was also assumed that client’s data vectors had entries such that at most 3 bytes are required to store the sum of up to all clients’ values without overflow.

Single-threaded simulations ran on a Linux workstation with an Intel Xeon CPU E5-1650 v3 (3.50 GHz), with 32 GB of RAM. Wall-clock running times and communication costs for clients are plotted in Figure 7.4. Wall clock running times for the server are plotted in Figure 7.5, with different lines representing different percentages of clients dropping out. Figure 7.6 shows wall-clock times per round for both the client and the server. Data transfer plots are omitted for the server, as these are essentially identical to those for the client, except higher by a factor of \( n \). This is because the incoming data of the server is exactly the total outgoing data of all clients, and vice versa. Bandwidth numbers for different numbers of dropouts are also omitted as the number of dropouts does not have a significant impact on this metric.

In the simulations, for both the client and the server almost all of the computation cost comes from expanding the various PRG seeds to mask the data vector. Compared to this, the computational costs of key agreement, secret sharing and reconstruction, and encrypting and decrypting messages between
clients, are essentially negligible, especially for large choices of $n$ and data vector size. This suggests that using an optimized PRG implementation could yield a significant running-time improvement over the prototype.

As seen in Figures 7.4a and 7.4b, the running time of each client increases linearly with both the total number of clients and the number of data vector entries, but does not change significantly when more clients drop out. In Figure 7.4c, the communication expansion factor for each client increases as the total number of clients increases, but this increase is relatively small compared to the impact of increasing the size of the data vector. This is also reflected in Figure 7.4d, where the communication expansion factor for each client increases as the total number of clients increases, but falls quickly as the size of the data vector increases. This shows that the cost of messages between clients amortizes well as the size of the data vector increases.

In the case of the server, Figures 7.5a and 7.5b show that the running time of the server increases significantly with the fraction of dropouts. This is because, for each dropped client $u$, the server must remove that client’s pairwise masks $p_{u,v}$ from each other surviving client $v$, which requires $(n-d)$ PRG expansions, where $d$ is the number of dropped users. In contrast, each undropped user entails only a single PRG expansion, to remove its self-mask. The high cost of dealing with dropped users is also reflected in the server running times in Figure 7.6.

The results of running the protocol over a Wide Area Network (WAN) are shown in Figure 7.7. The server and clients were run on geographically separated datacenters, with contention for CPU and network. This contention is reflected in the running times and client failures, which were caused by evictions from higher-priority jobs. The clients have a somewhat shorter runtime than the server: this is because the server has to run the additional, expensive unmasking step after all clients have completed.

### 7.7 Chapter Acknowledgements

The work presented in this chapter was joint work with Keith Bonawtiz, Vladimir Ivanov, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Karn Seth, and Aaron Segal. This work was done at Google.
Chapter 8

Conclusion

This thesis presented novel engineering techniques for MPC protocols, especially two-party protocols, in several settings. Early theoretical results established the feasibility of the various protocols presented in the preceding chapters, as well as a general framework for reasoning about security. On the other hand, real-world constraints continue to demand novel engineering techniques, on a case-by-case basis, for MPC deployments.

Chapters 3 and 4 describe techniques for garbled circuit systems. Previous work had assumed that circuits were the proper computation model for garbled circuit protocols, but as the work on BillionYao and PCF progressed, it became clear that the circuit model itself scales poorly, and that a “distributed CPU” model is better. Since presenting that work, others have presented further improvements by combining garbled circuit and oblivious RAM constructions, up to the point of creating a two-party MIPS virtual machine [WGMK16]. The high-level approach of PCF was applied to verifiable computation in Chapter 5, with an even greater improvement, as control-flow information is not secret in that setting.

Unfortunately, despite many engineering improvements garble circuits frameworks and performance improvements in garbled circuits protocols, the fact that communication grows with computation makes garbled circuits unacceptable for the two-party application presented in Chapter 6. Techniques for designing custom protocols remain an important area of research for real-world applications. An interesting future direction is the combination of customer protocols with garbled circuits; for example, using a customer PSI protocol to reduce the circuit size, followed by a garbled circuit that computes some functionality over the intersection. Another future direction would be to substitute Paillier encryption for a lattice-based construction, which would reduce the CPU cost in the case of sums and allow more general functions.

The application described in Chapter 7 involves more than two parties. Again, communication is the bottleneck and generic protocols are immediately ruled out in that setting. On the other hand, the use of linear secret sharing and the threshold requirements for security are similar in appearance to the BGW protocol [BOGW88]. Properly handling malicious clients without introducing a requirement for non-colluding servers remains an open problem; techniques involving zero-knowledge proofs appear to require far too much communication. One technical issue with validating client inputs is the fact that the clients must add many PRG streams, so any solution would likely require a different approach to protecting the privacy of client inputs.
Bibliography


[Wil] Zooko Wilcox. The design of the ceremony.


