Practical Secure Two-Party Computation

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Yan Huang

August 2012

Abstract

Secure two-party computation allows two parties to cooperatively evaluate a function that takes both parties' private data as input without revealing any private information other than the outcome. Although secure computation has many important applications in various fields and a general theoretical solution has been known for decades, practical systems are rare due to the prohibitive performance overhead and the tremendous effort required to build such systems.

The garbled circuit (i.e., Yao's circuit) technique enables secure computing of polynomial-time computable functions but has previously been thought to be too expensive for practical applications. We improved the efficiency of garbled circuits focusing on both execution and design aspects. Our implementation uses pipelining aggressively so the circuit execution runs with a nearly constant amount of memory and can scale to arbitrarily large circuits. To aid the efficient construction of circuits, we developed a library of component circuits that enables programmers to quickly create new ones by modular composition of existing ones. We integrated these ideas into a new framework that enables programmers to develope secure computation protocols from an existing insecure implementation while providing enough control over the circuit design to enable efficient implementation. To evaluate the effectiveness of our techniques and our new tools, we build several privacy-preserving applications which are secure against passive adversaries, including secure biometric identification, secure edit distance and Smith-Waterman, private encryption, and private set intersection.

The secure guarantees of passively-secure protocols do not hold if an attacker goes "active" — by deviating from the protocol specification. To thwart active adversaries, we present a concrete design and implementation of protocols achieving security guarantees that are much stronger than are possible with passively-secure protocols, at minimal extra cost. We consider protocols in which a malicious adversary may learn a single (arbitrary) bit of additional information about the honest party's input. Correctness of the honest party's output is still guaranteed. Adapting prior work of Mohassel and Franklin, the basic idea in our protocols is to conduct two separate runs of a (specific) semi-honest, garbled-circuit protocol, with the parties swapping roles, followed by an inexpensive secure equality test. We provide a rigorous definition and prove that this protocol leaks no more than one additional bit against a malicious adversary. In addition, we propose some

heuristic enhancements to reduce the overall information a cheating adversary learns. Our experiments show that protocols meeting this security level can be implemented at a cost very close to protocols that only achieve semi-honest security. Our results indicate that this model enables the large-scale, practical applications possible within the semi-honest security model, while providing stronger security guarantees.

We also explore the commodity-based paradigm for generic secure two-party computation. By trusting a third-party, not with private inputs, but only to provide correlated random numbers we can achieve a very low computation and communication overhead in both semi-honest and malicious threat models. The efficiency gains require a series of optimization techniques including layered circuit execution, round packing, traffic packing, and specialized circuit optimization for minimizing the cost of network latencies. Our experiments show that commodity-based protocols can be an order of magnitude more efficient (in both time and bandwidth) than the best known garbled circuit based ones assuming semi-honest adversaries. In presence of malicious adversaries, our approach offers even larger performance gains (more than 600x faster and 2500x more bandwidth-efficient compared to the state-of-art maliciously secure protocol).

Our results demonstrate that secure computation can be much more efficient than previously thought, and can scale to support large and interesting applications.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Yan Huang

This dissertation has been read and approved by the Examining Committee:

Westley Weimer, Committee Chair

David Evans, Adviser

Jonathan Katz

Aaron Mackey

Gabriel Robins

---

abhi shelat

Accepted for the School of Engineering and Applied Science:

James Aylor, Dean, School of Engineering and Applied Science

August 2012

To all who helped me survive my graduate school.

遊子吟

孟郊

慈母手中線，
遊子身上衣。
臨行密密縫，
意恐遲遲歸。
誰言寸草心，
報得三春暉。[1]

當你快樂時，你要想，這快樂不是永恆的；
當你痛苦時，你要想，這痛苦也不是永恆的。[2]

[1] *Traveler's Song*: the threads in the hands of a loving mother; the clothes on her travelling son; close and careful stitches right before departure; (She) fears her son could return late; (even) the so-called little inch-long grasses have ambitions; they meant to repay three days of spring sunshine.

[2] When you feel happy, keep in mind the happiness won't be eternal; when you suffer, keep in mind the suffering won't be eternal, either.

Acknowledgments

Wading through graduate school successfully with a Ph.D degree is indeed a long, tough process that attests one's determination, non-stopping hard work, and luck! I have been very fortunate to be advised by Prof. David Evans, who kept me well-funded throughout and generously offered me conference trips every year. His encouragements sustained me through the hard times of the graduate study. I am also lucky to be mentored by Prof. Jonathan Katz, whose insightful comments from time to time pulled me out of deep confusion. His brilliance saved me a great deal via throttling many of my dangerous and tricky research ideas. In addition, I feel fortunate to come across and work with a group of interesting people, including (but not limited to) Dr. Lior Malka (thank you for helping me embark on secure computation projects), Jeffrey Shirley (I remember all those interesting discussions with you and your encouragements during hard days), Peter Chapman, Samee Al Islam, and Chih-hao Shen. I enjoyed staying with the whole security research group at UVa over the years. It feels like a big family, especially after we got everyone sitting in that cozy, big office in Rice Hall. Last but not least, I appreciate the lovely pleasant atmosphere that UVa computer science department have nurtured for the graduate students.

The completion of this thesis beholds the endless love and support from my parents, who are more than seven thousand miles away from me (nevertheless, physical distance never separates this family).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of *secure two-party computation* is to enable two parties to cooperatively evaluate a function that takes private data from both parties as input without exposing any of the private data. At the end of the computation, the participants learn nothing more than the output of the function. Secure computation has many important applications such as privacy-preserving biometric identification, set intersection, and personal genetics. Theoretical secure computation solutions have been known since the 1980s, but real systems are scarce due to the high runtime costs associated with traditional techniques and the effort required to build them. The goal of my research is to make privacy-preserving computation practical enough that it can be used routinely in important, large-scale applications.

1.1 Motivating Applications

Motivating applications for two-party secure computation have three properties: (1) the application involves inputs from two independent parties; (2) each party wants to keep its own data secret; and (3) the participants agree to reveal the output of the computation. That is, the result itself does not imply too much information about either party's private input.

Many compelling applications have these properties. This section introduces four examples we will use in our evaluation: private biometric identification (Section 1.1.1), privacy-preserving genomics (Section 1.1.2), privacy-preserving encryption (Section 1.1.3), and private set intersection (Section 1.1.4).

1.1.1 Private Biometric Identification

Matching biometric data is critical to many identification systems including fingerprint- and face-recognition systems widely used in law enforcement. Such systems typically consist of a server-side database that holds a

set of biometric readings (stored electronically as feature records), and clients who submit candidate biometric readings to the server for identification. Typically, an identity match is signified by comparing the biometric data corresponding to some identity profile in the database and that from a client, with respect to some metric (e.g., Euclidean distance, or Hamming distance), assuming it is "close enough" (specifically, within some distance parameter ϵ).

The goal of *privacy-preserving* biometric identification is to enable biometric identification of the sort described above without revealing any information about the client's biometric data to the server, and without disclosing anything about the database to the client (other than the closest match, if within distance ϵ, or the non-existence of any close match).

Private biometric identification has applications in enabling collaborative criminal suspects searching between law-enforcement agents from different jurisdictions. For example, agents (or investigators) from different organizations might want to carry out some biometric identifcation process where one party holds the database while the other has the search key. Although jurisdictional complications may make it difficult for them to simply share the secret inputs upfront, it would be desirable to share the identification results (e.g., when the results are criminal suspects).

Fingerprint recognition. *Fingerprint recognition* (or *fingerprint identification*) is the task of searching for the *best* match in a database of fingerprints with a given candidate fingerprint. In contrast, *fingerprint authentication* seeks to determine if a candidate fingerprint matches a particular registered fingerprint. Techniques for matching fingerprints have been extensively studied. Maltoni et al. [75] provides more comprehensive information.

Depending on the sensing technology, fingerprint images exhibit traits at different levels of image quality. At the global level, ridge-lines shapes fall into one of several patterns such as *loop*, *whorl*, and *arch*. At the local level, there are about 150 different types of local ridge characteristics. At an even finer level, intra-ridge details are identified and used in high-end fingerprint applications. In the last decade, many fingerprint-recognition techniques have been developed that combine various features of the fingerprint [7, 87, 90, 106]. Most of them involve sophisticated training and classification algorithms, which are not suitable for developing an efficient privacy-preserving fingerprint recognition system. In contrast, the filterbank-based fingerprint matching algorithm proposed by Jain et al. [56] (also used by Barni et al. [5]) allows each party to independently compute feature vectors (i.e., FingerCodes) from its own fingerprint images without cooperation. Then a match is simply defined by the closest vector measured with Euclidean distance. This approach provides good accuracy and leads to an efficient privacy-preserving protocol because only the calculation of the distances

and a global minimum needs to be done securely.

Face recognition. Face recognition serves a another important application in biometric identification with many established algorithms. Common representation of faces include eigen vectors (e.g., Eigenfaces [100]) and bit strings [24, 84]. An Eigenface-based privacy-preserving face recognition prototype looks similar to the FingerCode-based secure fingerprint recognition system described above, but is much more expensive because the computation of robust feature patterns require substantial interaction which use both parties' secret eigen vectors [35]. In comparison, Osadchy et al. [84] show a face recognition prototype based on Hamming distance of feature bit strings that is both illumination- and orientation-robust and fits into privacy-preserving computation very well.

1.1.2 Private Genomics

Human genome sequences, which potentially explain one's vulnerabilities to genetic diseases, reveal character traits, and allow individual and family identification, are usually regarded as highly sensitive information. On the other hand, emerging and anticipated scientific advances on health and medicine depends heavily on computational analysis of such sensitive genomic data. In many scenarios, the computation needs to take sensitive inputs from multiple principals among which it could be cumbersome to establish full mutual trust. Secure computation technique can help to resolve this dilemma.

It is important and useful that patients of a certain disease share their experiences in fighting against the disease. They might want to share experiences on medicines or therapies by somehow releasing the improvements (or side-effects) brought by them, along with part of their genome sequences. Future patients could estimate the effects of different treatments by comparing their own genome sequences with those of others who have already taken them. Generic secure computation techniques allows this to be done without patients revealing their genomic information. Such medical applications may be a few years off, but we study the design and implementation of two sequence comparison algorithms (edit distance and Smith-Waterman [98]) that form the basis for many current and envisioned genomic applications. In addition, due to their importance, secure implementations of them were previously attempted, which makes it easier for performance comparison to evaluate the effectiveness of our general techniques. For example, Jha et al. explored implementing these protocols using generic secure computation protocols [61]. In Section 4.3 and 4.4 we show that the optimization techniques presented in this thesis bring an order of magnitude speedup than their best implementation.

1.1.3 Private AES

AES is a standard block cipher commonly used to construct symmetric encryption schemes. A privacy-preserving AES cipher allows Alice, who has a private key, to encrypt a private message from Bob without Alice knowing the message, nor Bob learning the private key. This primitive has a number of interesting applications. For example, it allows Bob, who provides a message, to ask Alice, who has a secret key, to sign the message *blindly* (without seeing the message). Such blind signatures could be used to authorize access to some encrypted data stored in the cloud (e.g., privacy-preserving search over encrypted data with a keyword encryption obtained securely).

Private AES is becoming a common benchmark for secure computation. Pinkas et al. [86] implements AES cipher as an SFDL program, which is in turn compiled to a huge SHDL circuit consisting of more than 30,000 gates. Henecka et al. used the same circuit, but obtained better online performance results by moving more of the computation to the precomputation phase. The best performance results they reported are 3.3 seconds in total and 0.4 seconds online per encryption cipher block [47]. Section 4.6 presents our approach of secure AES encryption that is more than an order of magnitude faster.

1.1.4 Private Set Intersection

Cryptographic protocols for *Private Set Intersection* (PSI) are the basis for many important privacy-preserving applications. It allows two parties holding sets S and S' to compute the intersection $I = S \cap S'$ without revealing to the other party any additional information about their respective sets (except their sizes). Either party, or both, may learn the intersection depending on the application. PSI can be used directly to enable two companies to find their common customers, or to allow a government agency to determine whether anyone on its terrorist watch list is present on a flight manifest. (Note that set intersection generalizes membership queries.) PSI can also be used as a sub-routine of larger privacy-preserving computations. For example, companies can perform data mining only on the customers they have in common (using PSI for pre-processing), or parties might apply some filter, privately specified by the other party, to their input set before computing the intersection (using PSI for post-processing). Many other examples are provided by De Cristofaro et al. [31]. Section 4.5 describes several of our constructions of the PSI protocol, each of which shows competitive performance in certain scenarios. We advocate generic approches in the semi-honest adversary model, especially when higher security settings are desirable.

1.2 Thesis

We argue that *the practicality of secure two-party computation can be substantially improved by employing better implementation techniques, adopting new (but realistic) threat models and by using alternative trust models.* We investigate a series of techniques to improve the efficiency and scalability of secure two-party computations. For example, the circuit execution is fully pipelined (where Boolean gates are processed in topological order) to enable the garbled circuit technique to work on large problems. Additionally, the framework we build enables the construction of efficient garbled circuits from optimized smaller circuits by better exploiting the free-XOR technique.

Furthermore, we explore alternative threat models whose performance is more realistic for practical scenarios, but without overly sacrificing security. We develop the k-leaked model proposed by Mohassel and Franklin [76], demonstrating that reasonable performance overhead (1.5 times that of semi-honest settings garbled circuit protocols) can be achieved against malicious adversaries by sacrificing a single extra bit of information. We also show that secure two-party computation can be one to many orders of magnitude faster than the state-of-art garbled circuit implementations as long as third parties exist to distribute properly constrained (private data independent) randomness.

1.3 Contributions

My research presents the following contributions:

1. Techniques that enable garbled circuit execution to scale to large applications. These techniques include pipelined execution, reducing circuit width, library-based circuit construction, and garbled/plain hybrid execution [Chapter 3]. They are built into an integrated software framework that facilitates creating secure two-party computing applications. In the semi-honest threat model, our framework produces protocols that are orders of magnitude faster than has been achieved in previous works [Section 3.4].
2. A concrete design and implementation of protocols in the 1-bit leakage threat model. This model offers much stronger security guarantees than are possible with semi-honest protocols, at minimal extra cost. Specifically, a malicious adversary may learn only a single bit of additional information about the honest party's input. The implementation features a highly efficient mechanism for carrying out the equality test in the presence of malicious adversaries, and incorporates pipelined execution and other efficiency optimizations [Chapter 5].
3. Developed a method for building efficient Boolean-circuit secure two-party computation protocols using commodity randomness [Chapter 6]. Protocols in this model can run one to many orders of

magnitude faster than garbled circuit based schemes, but without overly trusting a third party. Our main contributions in this work are effective solutions (including means to optimize circuits for depth and layered circuit execution) to mitigate the performance bottleneck caused by network latencies.

4. Evaluated the effectiveness of our tools by building several privacy-preserving applications, including genomic analysis, Hamming distance, Euclidean distance, AES, and set intersection [Chapter 4]. In the semi-honest threat model, we demonstrate secure computation of circuits with over 10^9 gates at a rate of roughly $10\mu s$ per garbled gate, which is order-of-magnitude improvement over the best previous implementations. In the 1-bit leak model, we show protocols with performance close to semi-honest settings but providing stronger security guarantees against malicious adversaries. Last, with server assistance for distributing correlated randomness, we show that private set intersection and AES can be computed orders of magnitude faster than the state-of-art maliciously secure protocol implementations.

Chapter 2

Background

This chapter provides necessary background on threat models, followed by general cryptographic primitives relevant to secure computation. Related works that are of interest with respect to certain techniques or specific applications are given later in the respective sections.

2.1 Threat Models

Threat models are used to capture the characteristics of an adversary's behavior. Depending on the restrictions over the adversary, existing models range from *semi-honest* (most restricted) to *fully malicious* (least restricted).

2.1.1 Semi-honest Model

The *semi-honest* (also known as *honest-but-curious*) threat model, assume that all parties follow the protocol as specified, but may attempt to learn additional information about the other party's input from the protocol transcript. Although it provides no guarantees if a party deviates the protocol, nor *fairness* (that both parties learn the output simultaneously), it is a standard security model for secure computation [40].

Studying protocols in the semi-honest setting is relevant for two reasons:

- There may be instances where a semi-honest threat model is appropriate: (1) when parties are legitimately trusted but are prevented from divulging information for legal reasons, or want to protect against future compromises; or (2) where it would be difficult for parties to change the software without being detected, either because software attestation is used or due to internal controls in place (for example, when parties represent corporations or government agencies).

- Protocols for the semi-honest setting are an important first step toward constructing protocols with stronger security guarantees. There exist generic ways of modifying the garbled-circuit approach to give covert security [3] or full security against malicious adversaries [68, 81, 70, 94].

Many interesting privacy-preserving applications do have the properties so that semi-honest garbled circuit protocols suffice. Namely, (1) both parties have a motivation to produce the correct result, and (2) only one party needs to receive the output. Examples include financial fraud detection (banks cooperate to detect fraudulent accounts), personalized medicine (a patient and drug company cooperate to determine the best treatment), and privacy-preserving face recognition. Chapter 5 explores a new technique for developing stronger protocols using semi-honest protocols as a building block.

2.1.2 Malicious Threat Model

It is usually unrealistic to assume passive adversaries who always obey the protocol specifications. To compromise the protocol security, an active adversary can deviate from the protocol in arbitrary ways, even at the risk of being caught cheating. Informally speaking, a two-party computation protocol is said to be secure in the malicious threat model if the privacy and correctness properties are guaranteed even in presence of such active adversaries.

With respect to a semi-honest garbled circuit based protocol, malicious adversaries could launch attacks in several (but not limited to such) ways. A malicious generator might construct a *faulty circuit* that discloses the evaluator's private input. For example, a circuit for f' of the adversaries choice (rather than the supposed f) is actually transmitted so that the victim's secret input could be revealed directly. In addition, more subtle attacks like *selective failure* exist [76]. In this attack, a malicious generator uses wire labels as inputs to the oblivious transfer that are inconsistent to those in garbled circuit construction. As a result, the evaluator's input can be inferred from whether the protocol execution completes successfully or not. For example, a cheating generator choosing (w^0, w^1) to be the pair of labels of an input wire of the garbled circuit could use $(w^0, \hat{w}^1)$, where $w^1 \neq \hat{w}^1$, in the corresponding oblivious transfer. Consequently, if the evaluator's input is 0, she will get w^0 from OT and complete the evaluation. In contrast, if her input is 1, she gets $\hat{w}^1$ and the execution will fail. We stress that, so long as the generator learns whether the protocol execution fails, the privacy leak persists even if it completes successfully.

Because there are an unlimited number of ways an active adversary can deviate from the protocol, showing particular attacks are impossible is insufficient to prove the whole protocol is secure against any active attacks. Thus, the convention to formally define security in the fully malicious model is by comparing two protocols executing in two different worlds (i.e., the *ideal* world and the *real* world, respectively) [40]. The ideal world

features the presence of a trusted third party who is delegated to receive secret inputs, run the computation locally, and distribute the results. In contrast, the two parties simply run the secure computation protocol in the real world in absence of any trusted party. A secure computation protocol Π is said to be secure if an adversary $\mathcal{A}$ corrupting a party in the real world obtains a distribution consisting of her view and the honest party's output, which is indistinguishable from the distribution of the ideal world outputs of both a probabilistic polynomial time simulator who corrupts the same party and the honest party. That is, no extra information is leaked by executing Π since whatever the adversary can learn (and affect) from a real world execution are all achievable by a polynomial time simulator running the ideal world protocol. We refer to Goldreich's classic book [40] on the detailed description of security in the fully malicious model.

The standard techniques to construct malicious adversary resistant protocols generally come in three different flavors.

Cut-and-choose The basic idea is to prepare many (e.g., 250) executions of the protocol, among which some (e.g., 2/5 of) traces are selected to verify the participants have followed the protocol while the rest are used for actual execution. The majority function is applied to the results from all actual runs to produce the final output. The scheme can be proved cryptographically secure against all probablistic polynomial time adversaries [68, 94].

Commit-and-prove The core idea, first suggested by Goldreich, Micali, and Widgerson [41], is to express every behavioral constraint in the protocol by an NP-language and prove that the constraints are satisfied using zero-knowledge proof of knowledge (ZKPoK) [42, 12]. Jarecki and Shmatikov [59] presented an approach where the generator is asked to prove the correctness of the garbled circuit in zero knowledge before the evaluation starts. Note that only a single copy of the circuit needs to be constructed. Nevertheless, such scheme is believed to be much less efficient than cut-and-choose because hundreds of expensive asymmetric cryptographic operations are needed per garbled gate.

MAC-then-compute Nielsen et al. [82] proposed a solution based on a technique called *authenticated bits*. Their key idea is to apply XOR-based message authentication code (MAC) to every bit throughout the collaborative computation, so that results remain authenticated only if both participants follow the protocol correctly. In other words, if a malicious participant deviates from the agreed protocol, the other party will notice and abort, but without risking any information leakage from the abortion. Hence, if the final result remains authenticated, it essentially proves that both parties behaved honestly. Although the protocol uses many expensive oblivious transfers (OT), an efficient OT extension protocol is devised to offset the cost via substituting expensive asymmetric operations with cheap symmetric

ones. Their solution shows good amortized efficiency. Chapter 6 presents an efficient commodity-based protocol adapted from this idea.

No matter what techniques are used to prevent protocol deviations in the original semi-honest protocol, we stress that all newly introduced behavioral constraints in the added mechanisms must also be enforced somehow. For example, if many copies of the circuit need to be evaluated (as in cut-and-choose), we have to guarantee that all evaluations are respecting the same inputs.

2.1.3 Covert Model

Protocols in the malicious threat model are too inefficient (orders of magnitude more expensive) compared to those in the semi-honest model, whereas the security guarantees offered by semi-honest protocols are too weak to fit in many scenarios. This dilemma leads to the search of some threat models in between. The most prominent of these is the covert model, introduced by Hazay, Aumann and Lindell [45, 3]. Aumann and Lindell et al. introduced the *covert* threat model [3]. In this model, a cheating adversary is "caught" with some constant probability, but with the remaining probability can (potentially) learn the honest party's entire input and arbitrarily bias the honest party's output. If an adversary is unwilling to take the risk of being caught, then such protocols will deter cheating altogether. Aumann and Lindell also show a two-party protocol with covert security that is only a small constant factor less efficient than the basic (semi-honest) garbled-circuit protocol.

Note that the 1-bit leakage model we consider in Chapter 5 is incomparable to the covert model. On the one hand, the single-bit leakage model allows the adversary to *always* learn one additional bit about the honest user's input, without any risk of being caught. On the other hand, the covert model allows the adversary to learn the entire input of the honest party with constant probability. The covert model also allows the adversary to affect the correctness of the honest party's output (with constant probability), something prevented in the single-bit leakage model.

2.2 Oblivious Transfer

One-out-of-two oblivious transfer (OT_1^2) [88, 36] is a crucial component of the garbled-circuit approach. An OT_1^2 protocol allows a *sender*, holding strings w^0, w^1, to transfer to a receiver, holding a selection bit b, exactly one of the inputs w^b; the receiver learns nothing about w^{1-b}, and the sender does not learn b. Oblivious transfer has been studied extensively, and several protocols are known. Naor and Pinkas [80] proposed an efficient OT_1^2 protocol based on Decisional Diffie-Hellman (DDH) hardness assumption that is secure in the

Input to SNDER: m pairs $\langle x_{i,0}, x_{i,1}\rangle$ of l-bit strings, where $1 \le i \le m$.
Input to RCVER: m selection bits $\boldsymbol{r} = [r_1, \cdots, r_m]$.

Protocol Output: RCVER outputs $\{x_{1,r_1}, x_{2,r_2}, \cdots x_{m,r_m}\}$ while knowing nothing of $\{x_{1,\overline{r_1}}, x_{2,\overline{r_2}}, \cdots x_{m,\overline{r_m}}\}$. SNDER learns nothing.

Preparation:

1. RCVER generates a $m \times k_1$ matrix T of random bits.
2. RCVER generates k_1 pairs $\langle \mathtt{key}_{i,0}, \mathtt{key}_{i,1}\rangle$ of k_2-bit strings, where $1 \le i \le k_1$.
3. SNDER generates a vector $\boldsymbol{s} = [s_1, \cdots, s_{k_1}]$ of random bits.
4. RCVER and SNDER execute Naor-Pinkas's OT protocol for k_1 times, where RCVER acts as the sender, SNDER as the receiver. At the i^{th} execution of OT_1^2, the message pair to send is $\langle \mathtt{key}_{i,0}, \mathtt{key}_{i,1}\rangle$, and the selection bit is s_i.

Execution:

SNDER		RCVER
		① RCVER prepares k_1 pairs $\mathtt{msg}_i = \langle \mathtt{msg}_{i,0}, \mathtt{msg}_{i,1}\rangle = \langle \mathsf{Enc}_{\mathtt{key}_{i,0}}(\boldsymbol{t}^i), \mathsf{Enc}_{\mathtt{key}_{i,1}}(\boldsymbol{r} \oplus \boldsymbol{t}^i)\rangle$, where $1 \le i \le k_1$.
	② $[\mathtt{msg}_1, \cdots, \mathtt{msg}_{k_1}]$. ←	
③ SNDER derives a bit matrix $\mathtt{Q}$, where $\boldsymbol{q}^i = \mathsf{Dec}_{\mathtt{key}_{i,s_i}}(\mathtt{msg}_{i,s_i})$. Then the SNDER prepares m pairs $\mathtt{msg}'_i = \langle \mathtt{msg}'_{i,0}, \mathtt{msg}'_{i,1}\rangle = \langle \mathsf{Enc}_{\boldsymbol{q}_i}(x_{i,0}), \mathsf{Enc}_{\boldsymbol{s}\oplus\boldsymbol{q}_i}(x_{i,1})\rangle$, where $1 \le i \le m$.		
	④ $[\mathtt{msg}'_1, \cdots, \mathtt{msg}'_m]$. →	⑤ $\forall 1 \le i \le m$, RCVER outputs $x_{i,r_i} = \mathsf{Dec}_{\boldsymbol{t}_i}(\mathtt{msg}'_{i,r_i})$.

Figure 2.1: The Oblivious Transfer Protocol

semi-honest setting. Based on the Random Oracle assumption, Ishai et al. [55] devised a technique to achieve a virtually unlimited number of oblivious transfers at the cost of (essentially) k executions of OT_1^2 (where k is a statistical security parameter) plus a marginal cost of a few symmetric-key operations per additional OT.

The two primitives can be combined to realize oblivious transfer of binary strings efficiently (Figure 2.1). Denote the i^{th} column vector of a matrix T by $\boldsymbol{t}^i$, and the i^{th} row vector of T by $\boldsymbol{t}_i$. The preparation phase can be done before any of the selection bits are known. At the end of the preparation phase, SNDER has k_1 keys $\mathtt{key}_{i,s_i}$, and RCVER has k_1 key pairs $\langle \mathtt{key}_{i,0}, \mathtt{key}_{i,1}\rangle$, where $(1 \le i \le k_1)$. These keys are later used to transmit the matrix $\mathtt{Q}$ efficiently. By using pre-computation, the on-line phase of our OT implementation requires only $2(k_1 + m)$ symmetric encryptions and $k_1 + m$ symmetric decryptions.

The correctness and security of this protocol follow directly from the proofs for Naor-Pinkas's OT (NPOT) protocol [80] and the extended OT protocol by Ishai et al. [55].

Correctness. The RCVER can learn x_{i,r_i} for all $1 \le i \le m$ following this case analysis:

1. If $r_i = 0$, then $\boldsymbol{q}_i = \boldsymbol{t}_i$ no matter what value s_i takes. Thus, RCVER knows the key $\boldsymbol{t}_i$, which is used to encrypt $x_{i,0}$.

2. When $r_i = 1$, the value of s_i selects whether $\boldsymbol{q}^i = \boldsymbol{t}^i$, or $\boldsymbol{r} \oplus \boldsymbol{t}^i$. However, this "selection" effect is canceled by xor-ing $\boldsymbol{s}$ and $\boldsymbol{q}_i$, so that it is always true that $\boldsymbol{s} \oplus \boldsymbol{q}_i = \boldsymbol{t}_i$, which is the key used to encrypt $x_{i,1}$.

Security. The security of our protocol follows from these two points:

1. The RCVER can never learn anything about $x_{i,\overline{r_i}}$ because it is encrypted using a different secret key which differs from that used for x_{i,r_i} by $\boldsymbol{s}$, the SNDER's random bit vector that is never revealed to the RCVER. The security property of NPOT used in the preparation phase guarantees that the selection bits of $\boldsymbol{s}$ are not revealed to RCVER.
2. In the first round of communication, either $\boldsymbol{t}^i$ or $\boldsymbol{r} \oplus \boldsymbol{t}^i$ is sent to the SNDER, but not both. Thus, the fact that the SNDER can never learn anything about the RCVER's selection bits $\boldsymbol{r}$ is derived directly from the security property of NPOT used in the preparation phase [80].

As a rough idea on the cost of oblivious transfer, the time for computing the "base" 80 oblivious transfers in semi-honest setting is about 0.6 seconds, while the on-line time for each additional OT_1^2 is about 15 μs.

We also note that there are known oblivious-transfer protocols with stronger security properties [46], as well as techniques for oblivious-transfer extension that are secure against malicious adversaries [44].

2.3 Garbled Circuits

Garbled circuits allow two parties holding inputs x and y, respectively, to evaluate an arbitrary function $f(x, y)$ without leaking any information about their inputs beyond what is implied by the function output. One party (the garbled-circuit *generator*) prepares an "encrypted" version of a circuit computing f; the second party (the garbled-circuit *evaluator*) then obliviously computes the output of the circuit without learning any intermediate values.

Starting with a Boolean circuit for f (which both parties fix in advance), the circuit generator associates two random cryptographic keys w_i^0, w_i^1 with each wire i of the circuit (w_i^0 encodes a 0-bit and w_i^1 encodes a 1-bit). Then, for each binary gate g of the circuit with input wires i, j and output wire k, the generator computes ciphertexts

$$\mathsf{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right)$$

for all inputs $b_i, b_j \in \{0, 1\}$. The resulting four ciphertexts, in random order, constitute a *garbled gate*. The collection of all garbled gates forms the garbled circuit that is sent to the evaluator. In addition, the generator reveals the mappings from output-wire keys to bits.

The evaluator must also obtain the appropriate keys (that is, the keys corresponding to each party's actual input) for the input wires. The generator (assuming the input wires corresponding to its n-bit input are numbered $1, \ldots, n$) can simply send $w_1^{x_1}, \ldots, w_n^{x_n}$, the keys that correspond to its own input where each $w_i^{x_i}$ corresponds to the generator's i^{th} input bit. The parties use *oblivious transfer* protocols to enable the evaluator to obliviously obtain the input-wire keys corresponding to its own inputs.

Given keys w_i, w_j associated with both input wires i, j of some garbled gate, the evaluator can compute a key for the output wire of that gate by decrypting the appropriate ciphertext. As described, this requires up to four decryptions per garbled gate, only one of which will succeed. Using the *point-and-permute* technique [74] (see Section 2.4.1), the construction can be modified so a single decryption suffices. Thus, given one key for each input wire of the circuit, the evaluator can compute a key for each output wire of the circuit. Given the mappings from output-wire keys to bits (provided by the generator), this allows the evaluator to compute the actual output of f. If desired, the evaluator can then send this output back to the circuit generator (note, however, that sending the output back to the generator can be a privacy risk unless the semi-honest model can be imposed through some other mechanism).

2.4 Implementation Techniques

Several notable works have improved aspects of secure function evaluation. We describe the most important ones, all of which are used in our framework, here.

2.4.1 Point-and-permute

A naïve implementation of the evaluator will decrypt every entry in the garbled truth table to find the correctly decryptable encryption. Malkhi et al. proposed the *point-and-permute* technique, which allows the circuit evaluator to identify the "right" entry in a garbled truth table to decrypt [74], saving the evaluator from decrypting more than one truth table entry. Take a unary gate as an example, we let s_{in} be the 1-bit semantic value on the input wire and denote with (w_{in}^0, w_{in}^1) the pair of labels associated with the input wire. The circuit generator selects a random *permute* bit p to associate with w_{in}^0 (which implies $\overline{p}$ is bound to w_{in}^1). The two encryptions in the garbled truth table will be fliped if and only if $p = 1$. Later, the value $p \oplus s_{in}$ is revealed to the evaluator to index (or *point* to) the "right" encryption to decrypt.

2.4.2 Free-XOR

The *free-XOR* technique [64, 63] allows all XOR gates be executed by just XOR-ing the input wire labels, without needing any encryption operations. The basic idea is for the circuit generator to keep a global

random bit string R such that for every wire only the label w^0 (representing 0) needs to be randomly sampled while the label w^1 (representing 1) is simply set to $w^0 \oplus R$. For every binary XOR gate (with input wires subscripted with i, j and the output wire with k), the label representing 0 on the output wire is derived from xor-ing corresponding input labels, i.e.,

$$w_k^0 = w_i^0 \oplus w_j^0.$$

With aforementioned setup,

$$w_k^1 = w_i^0 \oplus w_j^1 = w_i^1 \oplus w_j^0 = w_i^0 \oplus w_j^0 \oplus R.$$

Thus, XOR gates can be realized by locally xor-ing the input wire labels without any expensive cryptographic encryptions. Security was initially proved using the Random Oracle model [13], but modified by Kolesnikov et al. to use the weaker correlation robustness assumption [64].

2.4.3 Garbled Row Reduction

Pinkas et al. [86] proposed the *Garbled Row Reduction* (GRR) technique, which reduces the size of a garbled table to three entries (saving 25% of network bandwidth) for all non-free gates and is composable the with free-XOR technique, assuming a cryptographic hash function H (e.g., SHA-256) is used for encryption. For example,

$$\mathsf{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right) = H(w_i^{b_i}, w_j^{b_j}, b_i, b_j) \oplus w_k^{g(b_i,b_j)},$$

where w_i, w_j are input wires, w_k is the output wire, and g is the binary function to garble. The generator can always choose $w_k^{g(0,0)} = H(w_i^0, w_j^0, 0, 0)$, such that the encryption corresponding to input signals $(0, 0)$, $\mathsf{Enc}^k_{w_i^0, w_j^0}\left(w_k^{g(0,0)}\right)$ is always a string of 0 bits. Hence, makes it unnecessary to transmit this entry of all 0s.

2.5 Programming Tools

During the past decade, many different types of tools for secure computation proliferated in both research areas of cryptography and programming languages. Generally they accept as input a program written in a Turing-complete programming language, which can be either imperative [74, 15, 47, 73, 83, 110, 108, 25, 26, 72] or descriptive [97]. The systems outputs can be boolean circuits [74, 15, 47], source code in a generic programming language [73, 110, 108, 25, 26, 72], or a running protocol [97]. Some systems [74, 15, 47, 97, 73] employed cryptographic primitives to enable secure computation, whereas others [110, 108, 25, 26, 72, 83] did not. Instead, they were designed as tools that use types or annotations to specify security properties

and apply various static analysis techniques to ensure the security requirements are met. More detailed description of these works are given below.

Silaghi developed a *constraint programming* style declarative language called SMC [97] and demonstrated it by solving several interesting problems such as anonymous scheduling [89] and stable matching [2]. The underlying cryptographic primitive is secret sharing based arithmetic circuit evaluation [16].

MacKenzie et al. [73] developed a compiler to automate the production of secure two-party computation protocols for a restricted but interesting category of computation, which are operations on the prime-order groups $\mathbb{Z}_q$ and $\mathbb{Z}_p$ where p, q are primes and $q|(p-1)$. Their implementation was built on secure arithmetic primitive protocols realized by *threshold cryptography* [33, 39]. Applications of their framework are restricted to converting certain types of cryptographic primitives for some special purposes (e. g., deriving signature schemes that enforce *co-sign*, or deriving distributed oblivious transfer protocols from ordinary ones). Compared to their work, ours targets arbitrary computation by garbled circuits, though employing threshold crypto-systems to speedup part of our circuit could be an interesting future exploration.

Malkhi and collaborators developed Fairplay [74], a compile-and-interpret framework that automates the production of secure two-party computation protocols from conventional ones. The main interface Fairplay exposes to programmers is a simple Algol-like programming language called SFDL that supports very limited primitive data types (`boolean`, sized `int` and `enumerate`), expressions (addition, subtraction, comparison, and boolean logic operations), and statements (non-recursive functions, branches, and constant number iterative loops). SFDL programs are compiled to monolithic digital circuits (stored as SHDL files), which are interpreted by the server/client runtime environments for protocol execution. FairplayMP [15] and TASTY [47] are two derivative works from Fairplay. FairplayMP extended the SFDL language to describe secure multi-party computations, and built its runtime engine on a circuit-based technique for multi-party computation [11].

TASTY [47] enhances the functionality and performance of Fairplay. It extended Fairplay's SFDL to allow the programmer to specify where in the digital circuit to integrate some *arithmetic* circuits (limited to addition and constant multiplication) that are realized by homomorphic encryption schemes. They also incorporated the free-XOR technique [64]. However, their approach still started from compiling SFDL programs, so the programmer does not have enough control over the circuit construction to minimize bit widths or make maximal use of free-XORs as is possible with our proposed approach. Scalability and performance is also limited because they do not employ pipelined circuit execution.

Nielsen and Schwartzbach [83] developed a high-level language SMCL for secure multi-party computation. They modeled the secure multi-party computation with a variable number of *clients* and a *conceptual server*. In SMCL, data and computations can be categorized into *public* (to everyone), *private* (to one particular

client) and *secret* (only appear in server scripts). Their focus is on using a compiler to enforce two security properties about branches (both paths are executed sequentially, and terminate with *no public side effects*), and verifying that all potential information leaks by the outcome are properly annotated. In contrast, we focus on improving the efficiency of the cryptographic part. Our users derive implementations of secure protocols from existing non-secure ones and are encouraged to work directly on certain circuits. Like Fairplay, SMCL only supported very limited data types (`int` and `bool`) and disallow general loops and recursion involving secret data.

Many language and analysis tools provide means to specify and check privacy requirements and trust relationship for program data [110, 108, 25, 26, 72]. For example, Jif/split [108] can automatically partition programs into a number of slices according to the annotated security types and trust labels. To ensure security, computations on a piece of data are moved to the *principal* host who owns the data. Secure program partitioning relies heavily on static information flow analysis [78, 91]. The security properties, as the *sources* of the information flow, are either specified with type systems [102] or marked by labels [79]. At the *sink* of the flow, potential information leaks or relaxed non-interference must be explicitly granted with *declassification* [92, 66]. However, these systems cannot do secure computation without trust being explicitly granted. They are not designed to use special cryptographic tools to resolve privacy conflicts.

Chapter 3

Improving Efficiency and Scalability

Two main approaches exist to construct protocols for secure computation. The first approach exploits specific properties of the function being computed to design special-purpose protocols that are, presumably, more efficient than those that would result from generic techniques. A disadvantage of this approach is that each function-specific protocol must be designed, implemented, and proved secure.

The second approach relies on completeness theorems [107, 41, 40] for secure computation to derive protocols for computing any function f starting from a representation of f. One such approach is the garbled circuits technique introduced by Yao [107], which produces a semi-honest protocol for any function f based on the Boolean-circuit representation of f. This generic approach to secure computation has traditionally been viewed as being of theoretical interest only since the protocols that result require several symmetric-key operations per gate of the circuit being executed and the circuit corresponding to even a very simple function can be quite large.

Beginning with Fairplay [74], several implementations of generic secure two-party computation have been developed in the past few years [71, 86, 47] and used to build privacy-preserving protocols for various functions (e.g., [61, 35, 93, 84, 52]). Fairplay and its successors demonstrated that Yao's technique could be implemented to run in a reasonable amount of time for small circuits (i.e., up to 4 million gates), but left the impression that generic protocols for secure computation could not scale to handle large circuits or input sizes or compete with special-purpose protocols for functions of practical interest. Indeed, some previous works have explicitly rejected garbled-circuit solutions due to memory exhaustion [61, 84].

We observe that design decisions made by Fairplay, and followed in subsequent work, led researchers to severely underestimate the applicability of generic secure computation. In this chapter, we describe techniques for improving the performance and scalability of garbled circuit execution.

[0]The contents in this chapter are based on paper "Faster Secure Two-Party Computation Using Garbled Circuits" [50].

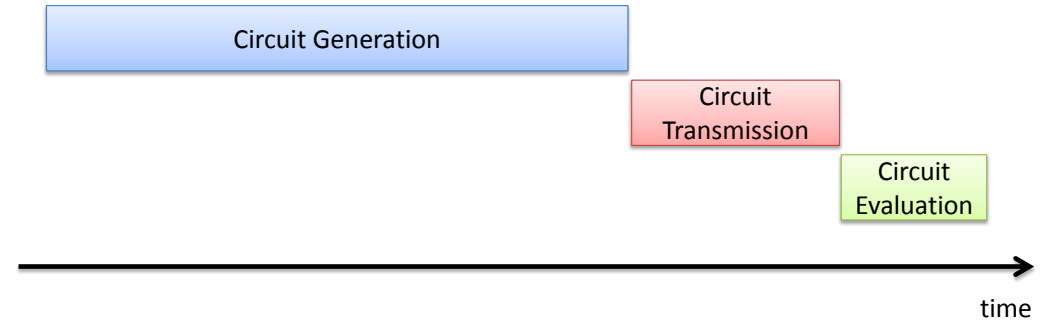
First, we find it not necessary to store the actual garbled circuits. Instead, the generator can generate the encryptions on the fly and send them to the evaluator over the network (respectively, the evaluator receiving the encryptions simply decrypts then discards them), leading to the idea of *pipelining* (Section 3.1). Then we study several ways to generate efficient secure computing protocols including library-based circuit construction (Section 3.2) and garbled circuit minimization (Section 3.3).

Our secure computation framework (Section 3.4), which combines the above ideas, allows programmers to construct protocols in a high-level language while providing enough control over the circuit design to enable efficient implementations. The framework has a small source code base, consisting of a main framework of about 1500 lines of Java code and a circuit library of an additional 700 lines. We describe the typical workflow of developing new secure computation protocols in Section 3.4.2.

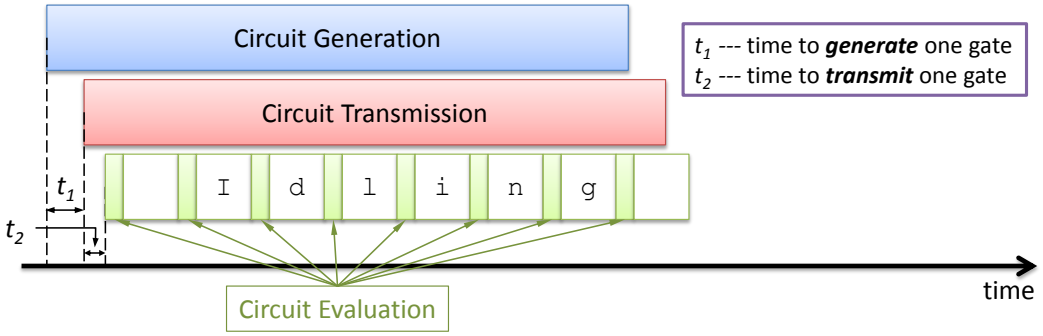
3.1 Pipelined Execution

The first limitation of previous garbled-circuit implementations (including Fairplay) is the memory required to store the entire circuit in memory, as illustrated in Figure 3.1(a). However, there is no need for either the circuit generator or evaluator to ever hold the entire circuit in memory. The circuit generation and evaluation processes can be overlapped in time (pipelined), eliminating the need to ever store the entire garbled circuit in memory as well as the need for the circuit generator to delay transmission until the entire garbled circuit is ready (Figure 3.1(b)). In our framework (Section 3.4), the processing of the garbled gates is pipelined to avoid the need to store the entire circuit and to improve the running time. As a result, garbled circuit execution becomes a nearly *constant* (as opposed to *linear* in prior implementations) space process. Pipelined execution is automated by our framework, so a user only needs to construct the desired circuit.

At the beginning of the execution, both the circuit generator and the circuit evaluator instantiate the circuit structure in exactly the same way. The Boolean circuit can be executed as normal in any topological order, as long as both the generator and the evaluator execute the circuit in exactly the same order (Figure 3.2). These design decisions do not cause any security problem because the circuit and its execution order do not depend on the private data. The order is public information shared between the two parties. When the protocol is executed, the generator transmits the garbled table (that is, three encrypted values) for every garbled gate (rather than the circuit structure) over the network, in a topological order defined by the circuit structure. As the client receives the encryptions, it is able to associate them with the corresponding gate of the circuit to do the evaluation. The overhead to keep the two parties synchronized is small, which is captured by the classic Producer-Consumer synchronization model.



(a) Non-pipelined Execution



(b) Pipelined Execution

Figure 3.1: Non-pipelined versus pipelined execution

Once a gate has been evaluated it is immediately discarded, so the number of garbled truth tables stored in memory is constant and small (exactly 3 encryptions). In addition, we stress that the publicly known circuit structures (not the actual garbled truth tables) can be reused iteratively for secure computation. Therefore, as long as the same set of basic circuit structures are used, evaluating more garbled gates does not increase the memory load on the generator or evaluator, but only affects the network bandwidth needed to transmit the garbled tables. Using larger or additional circuit structures does increase the memory requirement negatively, because more memory is needed to store the wires and connections. We will see in Section 3.2 how this issue can be alleviated by exploiting the modular and recursive nature of computation.

Apparently the pipelined execution only saves approximately half of the time compared to non-pipelined implementation. However, the side effects of pipelining on performance can also be substantial: (1) it eliminates the delay of accessing long-term storage (e.g., disk files); (2) it features stronger recursive pattern at run-time which leads to better program locality and cache misses.

3.2 Library-based Construction

Using pipelined execution, the memory cost of garbled circuits now grows only with the size of the (reusable) circuit structures. But, the execution time is still linear in the number of gates in the circuit. In particular,

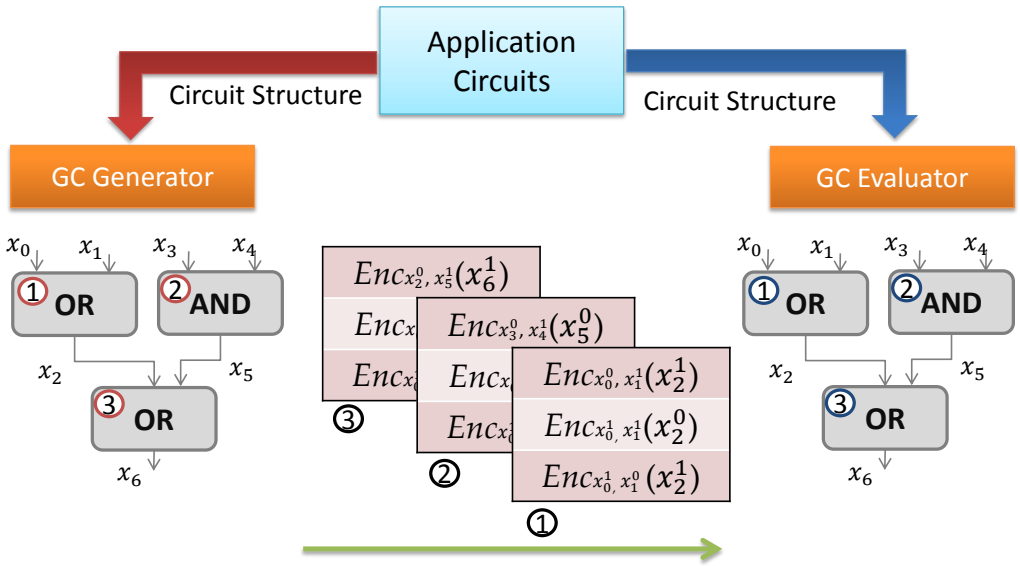


Figure 3.2: Pipelining synchronized on circuit structure and execution order

the cost of evaluating a garbled circuit protocol scales linearly in the number of non-XOR gates since we use the free-XOR technique (Section 2.4.2) to perform XOR without needing any cryptographic operations. Hence, to improve the execution efficiency, it is important to construct functionally equivalent circuits that use as few number of non-XOR gates as possible.

We advocate building large secure computation applications by composing smaller circuit components. The benefits of using existing library circuits are many-fold:

1. Always constructing everything from scratch (i.e., Boolean gates) is an awkward way to write medium to large scale applications. It easily becomes a nightmare for the developers to manage the complexity of software development. Making big circuits out of smaller ones enables us to exploit *software modularity*. In addition, modular circuit construction also encourages program *re-use*. For example, a comparison circuit such as `GT` will be useful both in finding global minimum, edit distance, and Sort-Compare-Shuffle based private set intersection. The `GT` circuit can thus be written and included in the library once for all.

2. For efficiency purposes, it is important to minimize the total number of non-XOR gates used in the garbled circuit realization of a certain functionality. However, such circuit optimization problem is known to be NP-hard in general (in terms of the size of circuit input and output) [19]. We would be likely to find optimal solutions for extremely small circuits such as a 1-bit adder. Based on the assumption that building large circuits from optimal smaller components should yield reasonable results, library-based circuit construction serves as a practical solution with reasonable performance.

3. Compared to high-level programming languages that support loops and recursive calls, Boolean circuits are extremely memory inefficient representations of computations. Although we never store any encryptions for any Boolean gate, the circuit size still matters because we have to maintain structures of all circuits. Library-based circuit construction allows us to represent computations efficiently by storing only a few circuit patterns instead of a big monolithic circuit. High-level language constructs like loops and recursive procedures can still be used to express complex computations, as long as control flows are never affected by private data or their derivatives. For example, during computing the Hamming distance of two bit-strings of a million bits, it suffices to keep just a single XOR gate in memory and invoke it a million times with a for-loop (rather than construct one circuit that has a million XOR gates in it), while only incrementing an oblivious counter of approximately 20 internal bits. This technique is fairly important to maintain reasonable performance when dealing with large scale secure computations.

We supply a library of basic circuits, including comparators, adders, muxers, minimizers. The basic circuits are parameterised with input sizes. For example, the adder class is defined by class ADD_2L_Lplus1, which adds up 2 ℓ-bit unsigned integers to place in an $(\ell + 1)$-bit integer; while the muxer is given by class MUX_2Lplus1_L, which chooses 1 out of 2 unsigned integers based on a 1-bit selection signal. An example showing usage of the basic building blocks is presented in Section 3.4.

3.3 Minimizing Secure Computation

Although the performance of garbled circuit execution can be substantially improved with the techniques above, it is still many orders of magnitude slower than native computation. So we only want to use expensive garbled circuit for computation that has to involve private data. In our approach, the applications are designed at the circuit level rather than using a high-level language like SFDL [74]. This enables us to take advantage of several opportunities for reducing the amount of secure computation that must be done.

3.3.1 Reducing Wires

Garbled circuits operate on bits, and every (non-XOR) Boolean bit operation requires expensive encryption operations. To improve performance, the circuits are constructed with the minimal width required for the correctness of the programs. This is achieved by designing most circuits with parameters that specify the sizes of the inputs. For example, SFDL's simplicity encourages programmers to count the number of 1s in a 900-bit number by writing code that leads to a circuit using 10-bit accumulators throughout the computation. In constrast, based on the insight that narrower accumulators will be sufficient at early stages, we can

use ten different accumulators, with input bit widths ranging from 1 to 10 bits to save unnecessary secure computation. Opportunity of savings by using dynamic bit width are also found in the secure edit distance protocol, where states in the upper-left part of the matrix need fewer input and output bits than those in the bottom-right. This technique has a significant impact on the overall efficiency — reducing the number of garbled gates needed for our edit distance protocol by 20%.

3.3.2 Static Propagation

In traditional garbled circuits, computations are done solely by encrypting and decrypting the wire labels which represent garbled signals. However, for many reasons, plain (un-garbled) Boolean signals are still available in garbled circuits and can even play a significant role in privacy-preserving computation. For example, as circuits are built upon their sub-component modules, some internal ports of the components need to be fixed to constant signals (e.g., the carry-in bit of an adder is fixed to 0). Moreover, when dealing with inputs that contain partial but dynamic secrets, a significant number of gates need not be executed in a garbled fashion. Therefore, we combine the plain execution with garbled execution dynamically, so that the former is used whenever possible. The garbed circuits implemented by our framework accept both plain and garbled signals and automate the hybrid execution. Hence, hybrid execution makes it easy to take a Hamming distance circuit (whose construction favors integer powers of 2) for two 1024-bit strings to produce one that works for inputs length of any integer (e.g., 777) between 512 and 1024, by simply assigning the extra input bits to `0`. Since all of the computation involving the fixed 0 inputs is *statically* done using plain execution, it will not incur any runtime overhead, leaving a cost nearly identical to what it would be for a custom-designed 777-bit width circuit.

3.3.3 Low-level Symbolic Execution

The wire labels obtained during a garbled circuit evaluation are normally treated as a worthless by-product of the evaluation, but can be used in subsequent computations. In the garbled circuit evaluator's perspective, the set of wire labels computed are meaningless numbers, conveying no semantic information until the last step. This property is bound to the rigorous definition of security for garbled circuit technique. We exploit this fact to avoid garbled execution for many binary gates.

Since wire labels are unique. We treat them as ordinary distinct symbols. It follows immediately from this observation that we can do binary gate level *symbolic execution* (means *generation* for the generator and *evaluation* for evaluator), which is essentially free compared to garbled execution. For example, let λ denote the value of a particular wire label. Symbolic execution rules are shown in the last two rows of Table 3.1. In

large circuits, these rules can help propagate the symbolic wire labels that collapse many binary gates to simple wire connections. In addition, combining the idea of hybrid and symbolic circuit execution, we can use the execution rules in the first two rows of Table 3.1.

Gate Type	AND ($\cdot$)	OR ($+$)	XOR ($\oplus$)
Symbolic Rules	$1 \cdot \lambda = \lambda$	$1 + \lambda = 1$	$1 \oplus \lambda = \overline{\lambda}$
	$0 \cdot \lambda = 0$	$0 + \lambda = \lambda$	$0 \oplus \lambda = \lambda$
	$\lambda \cdot \lambda = \lambda$	$\lambda + \lambda = \lambda$	$\lambda \oplus \lambda = 0$
	$\overline{\lambda} \cdot \lambda = 0$	$\overline{\lambda} + \lambda = 1$	$\overline{\lambda} \oplus \lambda = 1$

Table 3.1: Symbolic execution rules for binary gates

Note that the symbolic rules require inverting wire labels to tell λ from $\bar{\lambda}$. To make this possible, every wire includes an associated 1-bit isInverted flag that indicates whether or not its signal has been inverted. By this method, a NOT gate can be implemented by flipping the isInverted flag without any expensive cryptographic operations. Since it is a public knowledge that it is a NOT gate, both parties will always be consistent in inverting their respective flags, but without any interaction with the labels. These facts assure both correctness and security of the technique.

3.3.4 Dividing Sensitive Computation

We note that it is usually overkill to implement an entire program with garbled circuits. Instead, part of the computation does not actually involve any private data at all. For example, in both the edit distance and Smith-Waterman protocols, the initialization part can be safely computed without garbled circuits. Occasionally, each party could have some computations that involves merely its own private input. Those computations could avoid expensive garbled circuit execution as well.

In addition, sometimes it is even possible to push some computation that originally requires collaboration towards one that can be done locally. Take the secure AES as an example, the client has the message while the server has the key. A naïve transformation of AES will require the key schedule to run obliviously on the client side since it uses the server's private input. However, this can also be done through running the key schedule locally on the server side but use oblivious transfers to send the schedule output. Even though more bits need to be obliviously transfered compared to the naïve scheme, the savings from avoiding garbled circuit still outweigh the cost of oblivious transfer. As another example, when implementing the sort-compare-shuffle type of PSI protocols (Section 4.5.4), we take advantage of the observation that each party is able to sort their inputs locally, such that only the merging needs to be done with garbled circuits.

Therefore, our framework is designed to make it easy for programmers to combine secure and plain computation in ways that allow applications to minimize the amount of expensive secure computation needed.

We encourage programmers to think carefully about which modules really need to be realized with secure computation and only realize those in garbled circuits.

3.4 Framework

We first present an overview of our framework design. Then we describe the general steps to use the framework to build efficient secure computation protocols.

3.4.1 Design

Our framework includes a library of circuits defined for efficient garbled execution. Applications can be built by composing these circuits, but more efficient implementations are usually possible when programmers define custom-designed circuits.

The hierarchy of circuits is organized following the *Composite* design pattern [38] with respect to the build() method. Circuits are constructed in a modular fashion, using Wire objects to connect them together. Figure 3.4.1 provides a UML class diagram of the core classes of our framework. The Wire and Circuit classes follow a variation of the *Observer* pattern, which offers a kind of publish/subscribe functionality [38]. The main difference is that when a wire w is *connected* to a circuit on port p (represented as a position index to the inputWires array of the circuit), all the observers of the port p automatically become observers of w.

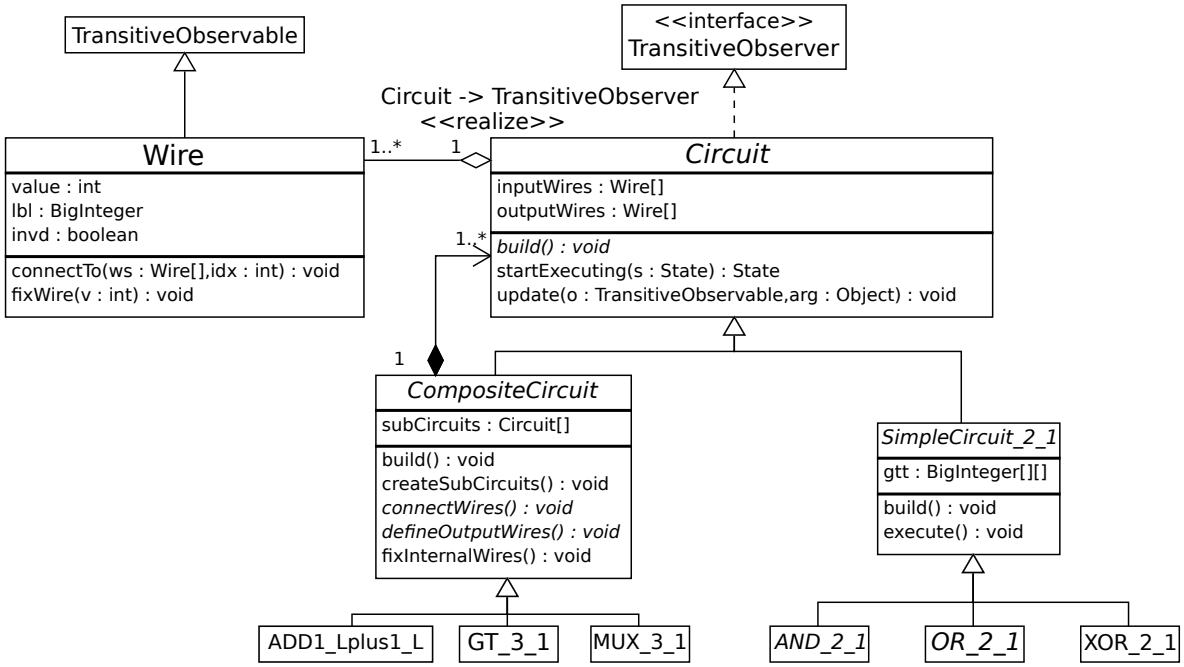


Figure 3.3: The core classes

The SimpleCircuit abstract class provides a library of commonly used functions starting with 2-to-1 AND, OR, and XOR gates, where the AND and OR gates are implemented using Yao's garbled-circuit technique and the XOR gate is implemented using the free-XOR optimization. Implementing a NOT gate is also free since it can be implemented as an XOR with constant 1.

The circuit library also provides more complex circuits for common operations including adders, muxers, comparators, min, max. These circuits are designed to minimize the number of non-XOR gates. Optimized circuits for additional functions can be added, as needed. A circuit for some desired function f can be constructed from the components provided in our circuit library, without needing to build the circuit entirely from AND/OR/NOT gates.

Composite circuits are constructed using the build() method, with the general structure shown below:

```
public void build() throws Exception {
    createInputWires();
    createSubCircuits();
    connectWires();
    defineOutputWires();
    fixInternalWires();
}
```

To define a new circuit, a user creates a new subclass of `CompositeCircuit`. Typically it is only necessary to override the `createSubCircuits()`, `connectWires()`, and `defineOutputWires()` methods. If internal wires are fixed to known values, these can be set by overriding `fixInternalWires()`. As mentioned in Section 3.3.2, our framework automatically propagates known signals which improves the run-time whenever any internal wires are fixed in this way. For example, given a circuit designed to compute the Hamming distance of two 1024-bit vectors, we can immediately obtain a circuit computing the Hamming distance of two 900-bit vectors by fixing 124 of each party's input wires to 0. Because of the way we do value propagation, this does not incur any runtime cost.

As a full example, the code for an AddOneBit circuit is given in Figure 3.4. The AddOneBit circuit, which takes as input an ℓ-bit number and a 1-bit signal and outputs an ℓ-bit integer, is simply built from an regular adder ADD_2L_Lplus1. Usually only four methods need to be rewritten, plus an optional `fixInternalWires()` function. The constructor simply calls its base class (`CompositeCircuit`)'s constructor, specifying that there are $m + 1$ input wires, m output wires, and one internal component circuit. Code in the base class will take care of initializing relevant resources. The component circuit, in this case, `ADD_2L_Lplus1`, gets instantiated in the `createSubCircuits()` method, which also invokes the corresponding method in its superclass to have

all of the component circuits initialized automatically. Next, with `connectWires()`, the interconnecting wires between the component circuits visible at the current circuit level (i.e., excluding the wire connections inside any component circuits) are programmed, using the `connectTo` method of the `Wire` class. In this case, the first wire is connected to the least significant input bit of the adder while the for-loop assigns the rest L bits as the first integer argument of the adder. In method `defineOutputWires`, the L output-wires of the AddOneBit is simply assigned by the L output wires of the internal adder. Last, since the most significant *L-1* bits of the second argument of the adder are all 0, we fix those wires with plain signal 0 in `fixInternalWires`.

```
class AddOneBit_Lplus1_L extends CompositeCircuit {
  private final int L;

  public AddOneBit_Lplus1_L(int m) {
    super(m + 1, m, 1);
    L = m;
  }

  protected void createSubCircuits() {
    subCircuits[0] = new ADD_2L_Lplus1(L);
    super.createSubCircuits();
  }

  protected void connectWires() {
    inputWires[0].connectTo(subCircuits[0].inputWires, 0);
    for (int i = 0; i < L; i++)
      inputWires[i+1].connectTo(subCircuits[0].inputWires, 2*i + 1);
  }

  protected void defineOutputWires() {
    System.arraycopy(subCircuits[0].outputWires, 0, outputWires, 0, L);
  }

  protected void fixInternalWires() {
    for (int i = 1; i < L; i++)
      subCircuits[0].inputWires[2*i].fixWire(0);
  }
}
```

Figure 3.4: Source code of AddOneBit circuit

3.4.2 Approach Workflow

The general workflow of our approach is illustrated in Figure 3.5. The development process can be started from an existing Java implementation of the targeted function. To build an efficient two-party secure computation protocol, a programmer first analyzes the target application to identify the critical components that need to be computed securely. Then, those components are translated to digital circuit designs. Some of the circuits

will be directly realized with existing circuits in the library, while others could require custom designs based on the basic modules in the library. Outputs of this transformation are Java classes. A few changes are also needed on the high level Java application to make sure it calls the garbled circuit implementation of the critical components. Finally, with support from our framework's core libraries, the circuits and the main program can be compiled and packaged into server-side and client-side programs that jointly instantiate the garbled-circuit protocol. They generator and evaluator are now ready to be deployed over a network, run as agents on behalf of two untrusted parties.

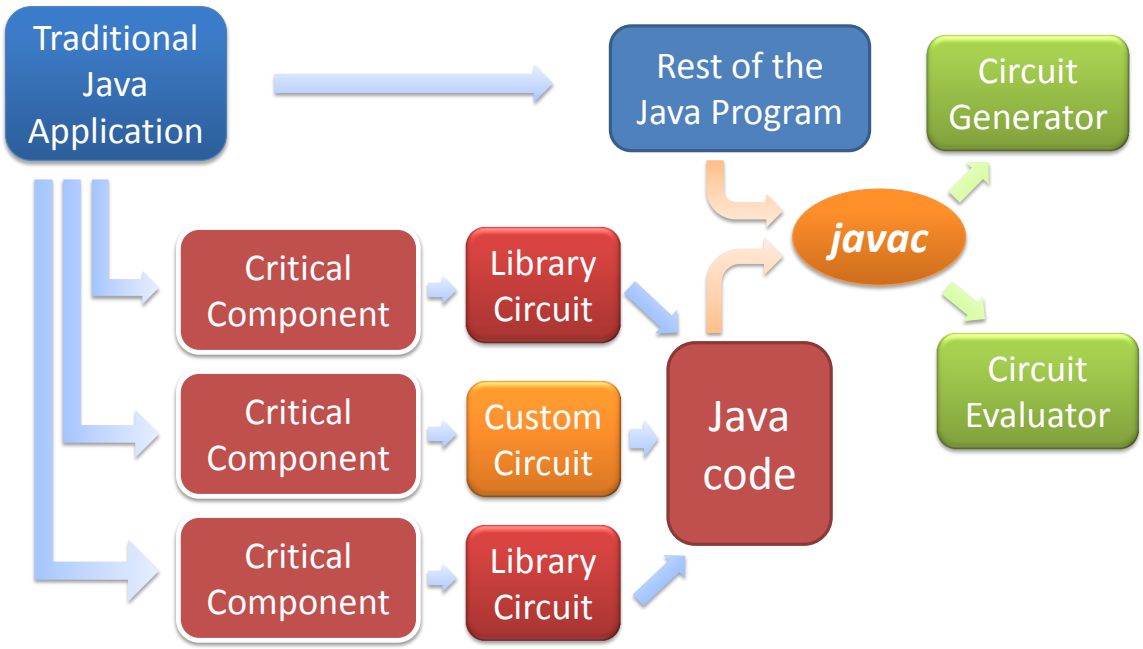


Figure 3.5: Workflow using the framework

Chapter 4

Case Studies

In this chapter, we study several example privacy-preserving applications to evalute the effectiveness of the general techniques presented in Chapter 3.

4.1 Experimental setup

Unless explicitly specified otherwise, the following settings are used throughout the experiments in our studies. We use 80-bit wire labels for garbled circuits and statistical security parameter $k = 80$ for oblivious-transfer extension. For the Naor-Pinkas oblivious-transfer protocol, we use an order-q subgroup of $\mathbb{Z}_p^*$ with $|q| = 128$ and $|p| = 1024$. These settings correspond roughly to the *ultra-short* security level as used in TASTY [47]. We used SHA-1 to generate the garbled truth-table entries. Each entry is computed as:

$$\mathsf{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right) = \text{SHA-1}\left(w_i^{b_i} \| w_j^{b_j} \| k\right) \oplus w_k^{g(b_i,b_j)}.$$

All cryptographic primitives were used as provided by the Java Cryptography Extension (JCE). Our experiments were performed on two Dell boxes (Intel Core Duo E8400 3GHz) connected on a local-area network.

4.2 Hamming Distance

Hamming distance is an essential metric with applications in myriad fields. It is a core operation in biometric identification systems [84] and m-point-SPIR (Symmetric Private Information Retrieval) [60]. Given two

[0]Section 4.2, 4.3, 4.4, and 4.6 are based on the paper "Faster Secure Two-Party Computation Using Garbled Circuits" [50]. Section 4.7 is based on parts of the paper "Efficient Privacy-Preserving Biometric Identification" [52]. Section 4.5 and 4.8 is based on the paper "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" [49].

ℓ-bit binary strings $\boldsymbol{a}$ and $\boldsymbol{b}$, where $\boldsymbol{a} = a_{\ell-1} \cdots a_1 a_0$ and $\boldsymbol{b} = b_{\ell-1} \cdots b_1 b_0$, the Hamming distance between $\boldsymbol{a}$ and $\boldsymbol{b}$, $Hamming(\boldsymbol{a}, \boldsymbol{b})$, is simply the total number of correspondingly different bits between $\boldsymbol{a}$ and $\boldsymbol{b}$. In a privacy-preserving scenario, $\boldsymbol{a}$ is the private input of Alice and $\boldsymbol{b}$ is the private input of Bob. Alice and Bob wish to collaboratively compute $Hamming(\boldsymbol{a}, \boldsymbol{b})$, or use its value as an intermediate result in a subsequent computation, without revealing their respective private inputs to the other.

4.2.1 Prior Work

Jarrous and Pinkas [60] proposed additive homomorphic encryption schemes for securely computing Hamming distance [60, 84]. Let $\boldsymbol{c} = c_{\ell-1} \cdots c_1 c_0$, where $c_i = a_i \oplus b_i$. Let $[\![c_i]\!]$ denote the encryption of bit c_i with Bob's public key pk_{P_2}. First, Alice computes $[\![c_i]\!]$ by computing $[\![a_i]\!] \cdot [\![b_i]\!]^{1-2a_i}$, where $[\![a_i]\!]$ is computed by Alice on her own and $[\![b_i]\!]$ is received from Bob. This works because,

$$\begin{aligned}[\![c_i]\!] = [\![a_i \oplus b_i]\!] = \left[\!\left[a_i \overline{b_i} + \overline{a_i} b_i\right]\!\right] = [\![a_i(1-b_i) + (1-a_i)b_i]\!] \\ = [\![a_i]\!] \cdot [\![b_i]\!]^{-a_i} \cdot [\![b_i]\!]^{1-a_i} = [\![a_i]\!] \cdot [\![b_i]\!]^{1-2a_i} .\end{aligned}$$

By homomorphically summing all c_i's, Bob obtains the encryption of $h = Hamming(\boldsymbol{a}, \boldsymbol{b})$:

$$[\![h]\!] = \left[\!\left[\sum_{0 \le i < \ell} c_i \right]\!\right] = \prod_{0 \le i < \ell} [\![c_i]\!] .$$

In practice, the value of the Hamming distance is rarely revealed directly. Instead, it is used obliviously in some subsequent computation. For example, in the SCiFI face-recognition protocol [84], h is compared to a threshold value to see if it signifies a close enough match. They accomplish this using a two step protocol. First, Alice computes $[\![h + r]\!]$ and sends it to Bob, where r is a random noise added to h to prevent Bob from learning h. Bob, who is able to decrypt $[\![h + r]\!]$ using his private key, only learns $(h + r)$. Second, Bob calculates whether h is below a threshold value t using a 1-out-of-$(h_{max} + 1)$ oblivious transfer protocol $\mathrm{OT}_1^{h_{max}+1}$, where h_{max} is the maximal possible value of h. For the $\mathrm{OT}_1^{h_{max}+1}$ oblivious transfer, Alice, who defines the threshold t, is the *sender*, with her $(h_{max} + 1)$-bit private input $\boldsymbol{x} = x_{h_{max}} \cdots x_1 x_0$, where

$$x_i = \begin{cases} 1, & \text{if } 0 \le (i - r) \bmod (h_{max} + 1) \le t; \\ 0, & \text{otherwise,} \end{cases}$$

while Bob is the *receiver*, using $(h + r) \bmod (h_{max} + 1)$ as his private input choice. At the end of the oblivious transfer, Alice learns nothing and Bob learns 1 if $0 \le h \le t$, or 0 otherwise. If the party (Bob in current

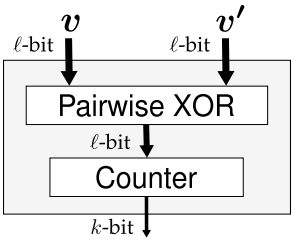


Figure 4.1: Hamming Distance Circuit.

settings) who doesn't define t is designated as the only receiver of the final outcome, a more expensive $\mathrm{OT}_1^{2d_{max}+1}$ protocol is required.

4.2.2 Circuit-based Approach

Hamming distance can be efficiently computed using garbled circuit protocols. The high level design of a Hamming distance circuit is given in Figure 4.1. It is basically an ℓ-way pair-wise XOR followed by a Counter circuit that counts the number of 1 signals among its input wires. The output of the Hamming circuit is a k-bit value, where $k = \lceil \log \ell \rceil$.

A naïve design of the Counter submodule is to use ℓ copies of a k-bit AddOneBit circuit, so that in each of the ℓ iterations the Counter circuit accumulates one bit of $\boldsymbol{v} \oplus \boldsymbol{v'}$ in the k-bit counter.

Since XOR gates are free and an k-bit Adder needs only k non-XOR gates [63], the Hamming circuit with the naïve Counter needs $\ell \cdot \lceil \log \ell \rceil$ non-free gates. We improve upon this by changing the Counter design to reduce the number of gates while enabling the gates to be evaluated in parallel.

First, we observe that the widths of the early one-bit adders can be far smaller than k bits. As mentioned in Section 3.2, our approach allows library circuits to be parameterized by bit width and circuits to be designed in a way that minimizes the number of bit operations needed. At the first level, the inputs are single bits, so a 1-bit adder with carry is sufficient; at the next level, the inputs are 2-bits, so a 2-bit adder is sufficient. This follows throughout the circuit, halving the total number of gates to $(\ell \lceil \log \ell \rceil)/2$.

Second, the serialized execution order is unnecessary. The naïve design can be improved to yield a parallel version of Counter given in Figure 4.2. Given that additional cores are available, the oblivious counting can be done in logarithmic time (in terms of the length of input bit string).

4.2.3 Experimental Results

Computing the Hamming distance between two 900-bit vectors took 0.019 seconds and used 56 KB bandwidth in the on-line phase (including garbled circuit generation and evaluation), with 0.051 seconds (of which the OT takes 0.018 seconds) spent on off-line preprocessing (including garbled circuit setup and the OT extension

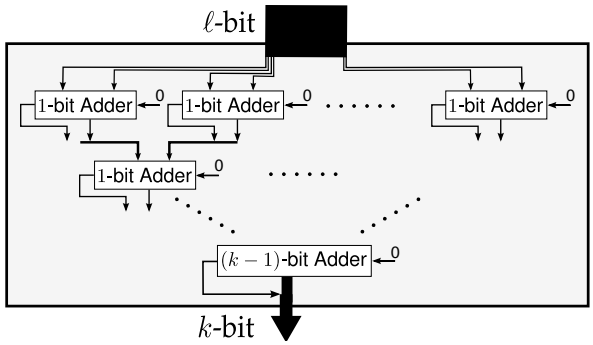


Figure 4.2: Parallelized Counter circuit.

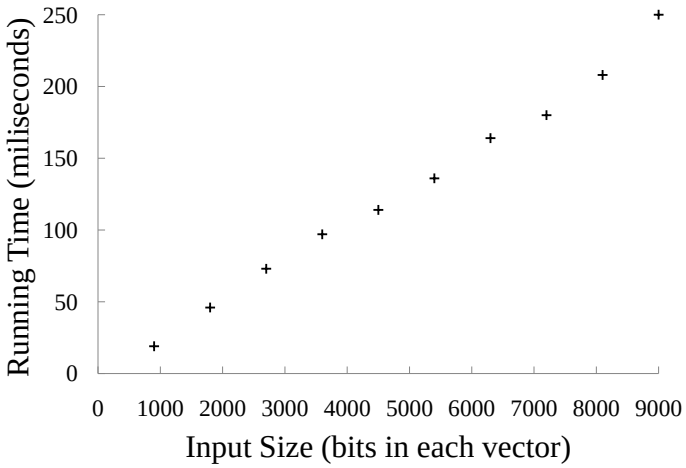


Figure 4.3: On-line running time of our Hamming-distance protocol for different input lengths.

protocol setup and execution). For the same problem, the protocol used in SCiFI took 0.31 seconds for on-line computation, even at the cost of 213 seconds spent on pre-processing.[1] The SCiFI paper did not report bandwidth consumption, but we conservatively estimate that their protocol would require at least 110 KB (comparing to 56 KB in ours). In addition to the dramatic improvement in performance, our approach is quite scalable. Figure 4.3 shows how the running time of our protocol scales with increasing input lengths.

The garbled-circuit implementation has another advantage compared to the homomorphic-encryption approach taken by SCiFI: if the obliviously calculated Hamming distances are not the final result, but are only intermediate results that are used as inputs to another computation, then a garbled-circuit protocol is much better in that by its nature it can be readily composed with any subsequent secure computation. The biometric matching application is an example scenario.

[1] Osadchy et al. [84] used a 2.8 GHz dual core Pentium D with 2 GB RAM for their experiments, so the comparison here is reasonably close. Also note that for their experiments, Osadchy et al. configured their host to turn off the Nagle ACK delay algorithm, which substantially improved network performance. This is not realistic for most network settings and was not done in our experiments.

4.3 Edit Distance

Edit distance (also known as *Levenshtein distance*) is a classic example for dynamic programming techniques [14]. It has applications in aligning DNA or protein sequences and comparing text files. Given two strings α and β, the edit distance between them (denoted $Levenshtein(\alpha, \beta)$) is defined as the minimum number of basic operations (add, delete, or replace a single character) needed to transform string α into β. The Levenshtein algorithm is given in Algorithm 1. This algorithm has the invariant that $\texttt{D[i][j]}$ always represents the Levenshtein distance between $\alpha[\texttt{1} \ldots \texttt{i}]$ and $\beta[\texttt{1} \ldots \texttt{j}]$. Lines 2–4 initialize each entry in the first row of the matrix $\texttt{D}$, while lines 5–8 initialize the first column. Within the two for-loops (lines 8–13), $\texttt{D[i][j]}$ is assigned at line 11 to be the smallest of three possible values, $\texttt{D[i} - \texttt{1][j]} + 1$, $\texttt{D[i][j} - \texttt{1]} + 1$, or $\texttt{D[i} - \texttt{1][j} - \texttt{1]} + \texttt{t}$ (where $\texttt{t}$ is 0 if $\alpha[\texttt{i}] = \beta[\texttt{j}]$ and 1 if they are different). This corresponds to the three basic operations: *insert* $\alpha[\texttt{i}]$, *delete* $\beta[\texttt{j}]$, and *replace* $\alpha[\texttt{i}]$ with $\beta[\texttt{j}]$, respectively.

Algorithm 1 *Levenshtein*(α, β)

```
 1: Initialize D[α.length][β.length];
 2: for i ← 0 to α.length do
 3:    D[i][0] ← i;
 4: end for
 5: for j ← 0 to β.length do
 6:    D[0][j] ← j;
 7: end for
 8: for i ← 1 to α.length do
 9:    for j ← 1 to β.length do
10:       t ← (α[i] = β[j]) ? 0 : 1;
11:       D[i][j] ← min(D[i − 1][j]+1, D[i][j − 1]+1, D[i − 1][j − 1]+t);
12:    end for
13: end for
```

4.3.1 Prior Work

Jha et al. gave the best previous implementation of a secure two-party protocol for computing the Levenshtein distance [61]. Instead of using Fairplay, they developed their own compiler based on Fairplay, while borrowing the function-description language (SFDL) and the circuit-description language (SHDL) directly from Fairplay. Jha et al. investigated three different strategies for securely computing the Levenshtein distance. Their first protocol (Protocol 1) directly instantiated Algorithm 1 as an SFDL program, which was then compiled into a garbled-circuit implementation. Because their garbled-circuit execution approach required keeping the entire circuit in memory, they concluded that garbled circuits could not scale to large inputs. The largest problem size their compiler and execution environment could handle before crashing was where the parties' inputs were 200-character strings over an 8-bit (256-character) alphabet.

Their second protocol combined garbled circuits with an approach based on *secure computation with shares*. The resulting protocol was scalable, but extremely slow. Finally, they proposed a hybrid protocol (Protocol 3) by combining the first two approaches to achieve better performance with scalability.

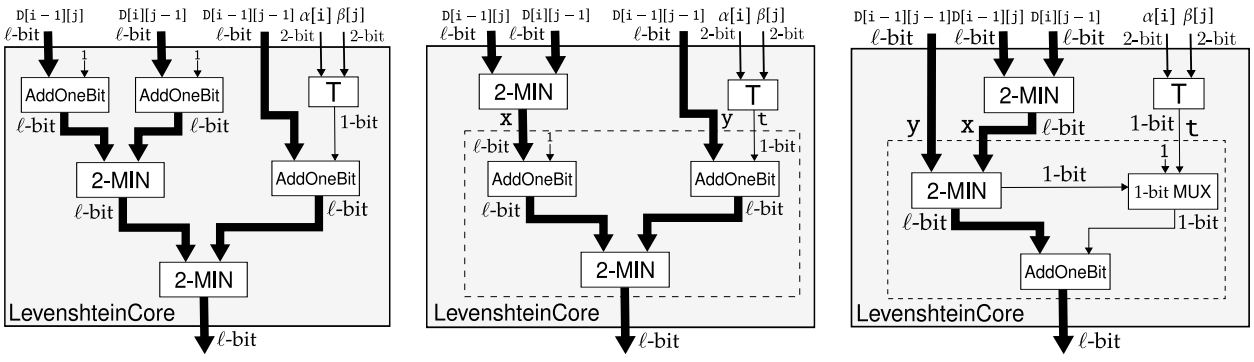
According to their results, it took 92 seconds for Protocol 1 to complete a problem of size 100×100 (i.e., two strings of length 100) over an 8-bit alphabet. This protocol required nearly 2 GB of memory to handle the 200×200 case [61]. Their flagship protocol (Protocol 3), which is faster for larger problem sizes, took 658 seconds and used 364.3 MB bandwidth on a problem of size 200×200 over an 8-bit alphabet.

4.3.2 Our Approach

First, we note the portion of the computation responsible for initializing the matrix (lines 2–7) does not require any collaboration, and thus can be completed by each party independently. Moreover, since the length of each party's private string is not meant to be kept secret, the two for-loops (lines 8–9) can be managed by each party independently as long as they keep the inner executions synchronized, leaving only two lines of code (lines 10–11) in the innermost loop that need to be computed securely.

Let ℓ denote the length of the parties' input strings, assumed to be over a σ-bit alphabet. Figure 4.4(a) presents a circuit, LevenshteinCore, that is computationally equivalent to lines 10–11 of Algorithm 1. The T (stands for "test") circuit in that figure outputs 1 if the input strings provided are different. Figure 4.5 shows the structure of the T circuit. (For the purposes of the figures in this section, we assume $\sigma = 2$ since this is the alphabet size that would be used for genomic comparisons. Nevertheless, everything generalizes easily to larger σ.) For a σ-bit alphabet, the T circuit uses $\sigma - 1$ non-free gates.

The rest of the circuit computes the minimum of the three possible edits (line 11 in Algorithm 1). We begin with the straightforward implementation shown in Figure 4.4(a). The values of D[i − 1][j], D[i][j − 1], and D[i − 1][j − 1] are each represented as ℓ-bit inputs to the circuit. For now, this is fixed as the maximum value



(a) Naïve (b) Better (c) Final

Figure 4.4: Implementations of the Levenshtein core circuit.

of any D[i][j] value. Later, we reduce this to the maximum value possible for a particular core component. Because of the way we define ℓ there is no need to worry about the carry output from the adders since ℓ is defined as the number of bits needed to represent the maximum output value. The circuit shown calculates exactly the same function as line 11 of Algorithm 1, producing the output value of D[i][j]. The full Levenshtein circuit has one LevenshteinCore component for each i and j value, connected to the appropriate inputs and producing the output value D[i][j]. The output value of the last LevenshteinCore component is the Levenshtein distance.

Recall that each ℓ-bit AddOneBit circuit uses ℓ non-free gates, and each ℓ-bit 2-MIN uses 2ℓ non-free gates. So, for problems on a σ-bit alphabet, each ℓ-bit NaiveLevenshteinCore circuit uses $7\ell + \sigma - 1$ non-free gates.

Two optimizations are possible that reduce the number of non-free gates involved in computing the Levenshtein core to $5\ell+\sigma$. First, since $\texttt{min}(\texttt{D[i}-1\texttt{][j}]+1, \texttt{D[i][j}-1\texttt{]}+1)$ is equivalent to $\texttt{min}(\texttt{D[i}-1\texttt{][j}], \texttt{D[i][j}-1\texttt{]})+1$, we can combine the two AddOneBit circuits (at the top left of Figure 4.4(a)) into a single one, and interchange it with the subsequent 2-MIN as shown in Figure 4.4(b). The circuits in the dashed box in Figure 4.4(b) compute $\texttt{min}(\texttt{x}+1, \texttt{y}+\texttt{t})$, where $\texttt{t} \in \{0, 1\}$. This is functionally equivalent to:

$$\textbf{if } (\texttt{y} > \texttt{x}) \textbf{ then } \texttt{x}+1 \textbf{ else } \texttt{y}+\texttt{t}.$$

Hence, we can reuse one of the AddOneBit circuits by putting it after the GT logic embedded in the MIN circuit. This leads to the optimized circuit design shown in Figure 4.4(c). Note that the 1-bit output wire connecting the 2-MIN and 1-bit MUX circuits is essentially the 1-bit output of the GT sub-circuit inside 2-MIN. This change reduces the number of gates in the core circuit to $2 \times 2\ell + \ell + \sigma - 1 + 1 = 5\ell + \sigma$.

The second optimization takes advantage of the observation that the minimal number of bits needed to represent D[i][j] varies throughout the computation. For example, one bit suffices to represent D[1][1] while more bits are required to represent D[i][j] for larger i's and j's. The value of D[i][j] can always be represented using $\lceil \log \min(\texttt{i}, \texttt{j}) \rceil$ bits. The number of gates decreases by:

$$1 - \frac{\sum_{i=1}^{\ell} \sum_{j=1}^{\ell} \left\lceil \log \left[\min(i, j) \right] \right\rceil}{\ell^2 \lceil \log \ell \rceil}.$$

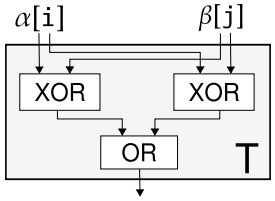


Figure 4.5: T circuit.

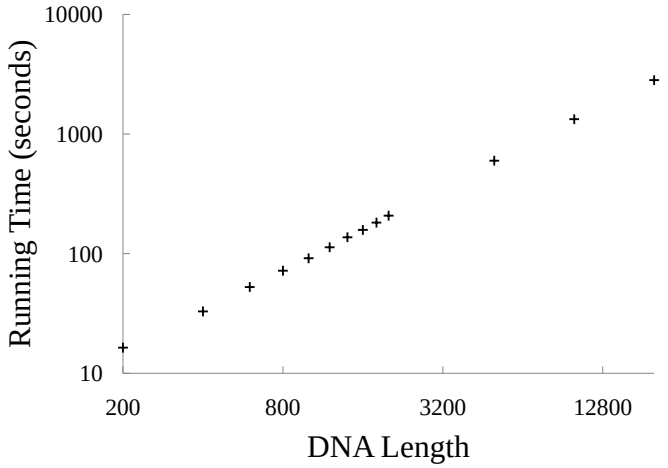


Figure 4.6: Overall running time of our Levenshtein-distance protocol. (Plotted on a log-log scale; the problem size is 200×DNA Length and $\sigma = 2$.)

For $\ell = 200$ this results in a 25% savings, but the effect decreases as ℓ grows.

4.3.3 Experimental Results

Our edit distance protocol handles arbitrary input lengths ℓ (it also handles the case where the input strings have different lengths) and arbitrary alphabet sizes 2^σ. It completes a problem of size 200×200 over a 4-character alphabet (enough for representing nucleotides in DNA) in 16.4 seconds (of which less than 1% is due to OT) using 49 MB bandwidth. The dependence of the running time on σ is small: for $\sigma = 8$ our protocol takes 18.4 seconds in the 200×200 case, which is 29 times faster than the results of Jha et al. [61].

Our protocol is highly scalable, as shown in Figure 4.6. The largest problem instance we ran is 2000×10000 (not shown in the figure), which used a total of 1.29 billion non-free binary gates and completed in under 223 minutes (at a rate of over 96,000 gates per second). In addition, our approach enables further optimizations for many practical scenarios. For example, if the parties are only interested in determining whether the Levenshtein distance is below some threshold d, then only the $\lceil \log d \rceil$ low-order bits of the result need to be computed and the number of bits for an entry can be reduced.

4.4 Smith-Waterman Score

The Smith-Waterman algorithm (Algorithm 2) is a popular method for genome and protein alignment [98, 77]. In contrast to edit distance which measures *dissimilarity*, the Smith-Waterman score measures *similarity* between two sequences (higher scores mean the sequences are more similar). The algorithm has a basic structure similar to the algorithm for computing edit distance. The differences are: (1) the preset entries

(the first row and the first column) are initialized to 0; (2) the algorithm has a more sophisticated core (lines 10–12) that involves an affine gap function `gap` and the maximum score of all previous entries in the row and column; and (3) the algorithm uses a fixed 2-dimensional score matrix `score` to determine the score given any 2 characters in the alphabet.

Algorithm 2 `Smith-Waterman`(α, β, `gap`, `score`)

```
1: Initialize D[α.length][β.length];
2: for i ← 0 to α.length do
3:    D[i][0] ← 0;
4: end for
5: for j ← 0 to β.length do
6:    D[0][j] ← 0;
7: end for
8: for i ← 1 to α.length do
9:    for j ← 1 to β.length do
10:      rMax ← max_{1≤o≤i}(D[i − o][j] + gap(o));
11:      cMax ← max_{1≤o≤j}(D[i][j − o] + gap(o));
12:      D[i][j] ← max(0, rMax, cMax, D[i − 1][j − 1] + score[α[i]][β[j]]);
13:   end for
14: end for
```

In practice, the gap function is typically of the form $\texttt{gap}(x) = a + b \cdot x$ where a, b are publicly known, negative integer constants. By choosing a and b appropriately, one can account for the fact that the evolutionary likelihood of inserting a single large DNA segment is much greater than the likelihood of multiple insertions of smaller segments (of the same total length). A typical gap function is $\texttt{gap}(x) = -12 - 7x$, which is what we use in our evaluation experiments.

The 2-dimensional score matrix `score` quantifies how well two symbols from an alphabet match each other. In comparing proteins, the symbols represent amino acids (one of twenty possible characters including stop symbols). The entries on the diagonal of the `score` matrix are larger and positive (since each symbol aligns well with itself), while all others are smaller and mostly negative numbers. The actual numbers vary, and are computed based on statistical analysis of a genome database. We use the BLOSUM62 [48] score matrix for computation over randomly generated protein sequences.

To obtain the optimal alignment, one first computes matrix `D` using Algorithm 2, then finds the entry in `D` with the maximum value and traces the path backwards to find how this value was derived. In a privacy-preserving setting, the full trace may reveal too much information. Instead, it may be used as an intermediate value for a continued secure computation, or just aspects of the result (e.g., the score or starting position) could be revealed.

The core of the Smith-Waterman algorithm (lines 10–12 of Algorithm 2) involves ADD and MAX circuits. To reduce the number of non-free gates, we replace lines 10–11 with the code in Algorithm 3. This allows us

Algorithm 3 Restructured Smith-Waterman core

```
rMax ← 0;
for o ← 1 to i do
  rMax ← max(rMax, D[i − o][j] + gap(o));
end for
cMax ← 0;
for o ← 1 to j do
  cMax ← max(cMax, D[i][j − o] + gap(o));
end for
```

to use much narrower ADD and MAX circuits for some entries since we know the value of D[i][j] is bounded by $\lceil \log(\min(i, j) \cdot maxscore) \rceil$, where *maxscore* is the greatest number in the score matrix. We only need to make sure that values are appropriately sign-extended (a free operation) when they are carried between circuits of different width.

Note that gap(o), which serves as the second operand to every ADD circuit, can always be safely computed without collaboration since it does not depend on any private input. Thus, instead of computing gap(o) using a complex garbled circuit, it can be computed directly with the output value fed directly into the ADD circuit. Being able to tightly bound the part of the computation that really needs to be done privately is another advantage of our approach (see Section 3.3).

The matrix-indexing operation on score does need to be done in a privacy-preserving way since its inputs reveal symbols in the private inputs of the parties. Since the row index and column index each can be denoted as a 5-bit number, we could view the score table as a 10-to-1 garbled circuit (whereas each entry in truth table is an encryption of 5 wire keys representing the output value). Using an extension of the *permute-and-encrypt* technique (Section 2.4.1), it leads to a garbled table containing $2^{10} = 1024$ ciphertexts (of which 624 are null entries since the actual table is 20×20, but which must be transmitted as random entires to avoid leaking information). However, observe that one of the two indexes is known to the circuit generator since it corresponds to the generator's input value at a known location. Hence, we use the index known to the circuit generator to specialize the two-dimensional score table lookup to a one-dimensional table lookup. This reduces the cost of oblivious table lookup to computing and transmitting 20 ciphertexts and 12 random entries (to fill the 2^5-entry table) for the circuit generator, while the work for the circuit evaluator is still performing one decryption.

4.4.1 Experimental Results

The secure Smith-Waterman protocol takes 415 seconds and generates 1.17 GB of network traffic running on two protein sequences of length 60. The garbled-circuit implementation by Jha et al. did not scale to a 60×60 input size, but their Protocol 3 was able to complete on this input length in nearly 1000 seconds

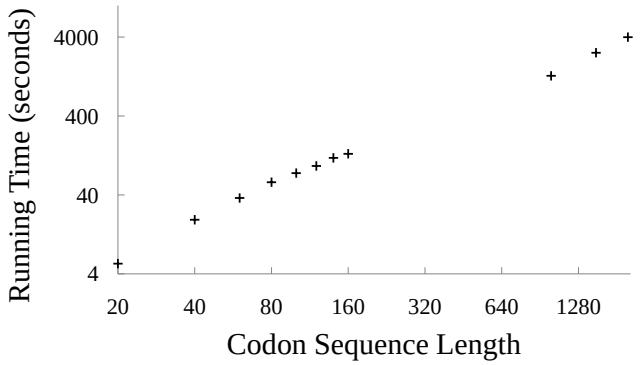


Figure 4.7: Overall running time of the Smith-Waterman protocol. (Plotted on a log-log scale; problem size 20×Codon Sequence Length.)

(but due to the simplifications they used, their implementation would not usually produce the correct result). Figure 4.7 shows the running time of our implementation as a function of the problem size.

4.5 Private Set Intersection

Private set intersection, introduced in Section 4.5, is an important building block for many interesting privacy-preserving applications. We assume two parties hold sets $S = \{s_1, s_2, \ldots, s_n\}$ and $S' = \{s'_1, s'_2, \ldots, s'_n\}$, respectively, where $s_i, s'_i \in \{0,1\}^\sigma$ and we assume neither S nor S' contains any duplicate elements. Each party's set is of (known) size n and all elements are exactly σ bits long (using padding it is easy to handle the case where set sizes are different or even kept hidden, up to a known upper bound, or where elements have different sizes). We also assume both sets change at each invocation (note that computing PSI repetitively with a static input set can cause substantial leakage merely by revealing the outputs). The goal is for the parties to compute the intersection $I = S \cap S'$ without revealing any information other than I.

4.5.1 Protocols Overview

Using garbled circuits, set intersections can be securely computed in several different approaches. We describe them as below, from the most simplistic to more sophisticated designs. Our first protocol (*Bitwise-AND* (BWA)), described in Section 4.5.2, uses a circuit based on a bit-vector representation of the parties' sets. The protocol is only practical for small universes; in that case, however, it achieves the best performance.

Section 4.5.3 describes the *Pairwise-Compare* (PWC) protocol that uses a circuit performing pairwise comparisons of the elements in the two parties' sets. This protocol has worst-case complexity $\Theta(n^2)$ for computing the intersection of two sets of size n, and is a reasonably good choice — even for large universes — as long as n is small. We present an optimization that improves performance when the size of the intersection

Protocol	Number of Non-Free Gates
Bitwise-AND (BWA)	2^σ
Pairwise-Comparisons (PWC)	$\big((2n-\hat{n})^2+\hat{n}\big)\,(\sigma-1)/4$
Sort-Compare-Shuffle-SORT	$2\sigma n\log(2n)+\big((3n-1)\sigma-n\big)+2\sigma n\log^2(2\hat{n})$
Sort-Compare-Shuffle-HE	$2\sigma n\log(2n)+\big((3n-1)\sigma-n\big)+(\sigma+32)n$
Sort-Compare-Shuffle-WN	$2\sigma n\log(2n)+\big((3n-1)\sigma-n\big)+\frac{\sigma(n\log n-n+1)}{3}$

Table 4.1: Gate counts for our protocols.
The size of each set is n, elements are represented using σ bits, and $\hat{n}$ is the size of the intersection.

is large without sacrificing any privacy. Though relatively simple, this and the previous protocol demonstrate that even straightforward approaches can produce effective PSI solutions using garbled circuits.

Section 4.5.4 presents our most involved protocols, which are all based on a *Sort-Compare-Shuffle* design. These protocols have complexity $\Theta(n\log n)$ with small constant factors. The main idea is for each party to sort their set locally, and then (privately) merge their sorted sets into a single sorted list. Then each adjacent pair of elements is compared (obliviously), with the value retained if the elements in the pair are equal, and a dummy value substituted otherwise. Finally, the resulting list of matching/dummy elements is obliviously shuffled before the entire list is revealed. This shuffling step is necessary because otherwise positional information about the matching elements leaks information about the non-matching elements in the parties' sets. We consider three different ways to perform the final oblivious shuffling: (1) obliviously sorting the entire list of matching/dummy elements using a garbled-circuit approach (SCS-SORT), (2) randomly shuffling the list of matching/dummy elements using a protocol based on homomorphic encryption (SCS-HE), and (3) randomly shuffling the list as before, but using garbled circuits applied to Waksman's oblivious switching network (SCS-WN).

Table 4.1 gives the costs of our protocols in terms of the number of gates that are garbled and evaluated, as a function of the size n of the input sets, the number of bits σ needed to represent each set element, and the size $\hat{n}$ of the intersection. XOR gates are not counted since these can be implemented "for free" (without performing any cryptographic operations) using the free-XOR optimization [64]. For the BWA and SCS-HE protocols, there are substantial other costs so gate counts alone do not capture the full cost of those protocols.

4.5.2 Bitwise-AND Protocol

The BWA protocol is designed for sets whose elements are drawn from a small universe. In this case, a set can be represented by a bit-vector of length 2^σ, and the set intersection can be computed simply by bit-wise AND-ing the bit-vectors of the two parties. The output is exactly a bit-vector representation of the intersection.

A circuit for this computation is straightforward, and is obtained by instantiating a binary AND gate 2^σ times. Although the cost of the resulting protocol grows exponentially with σ, the small constant factor involved leads to good performance when σ is small. Indeed, for values of σ up to 16, we found this to be the most efficient protocol in our experiments.

The BWA protocol does not restrict the size of the parties' sets, so a dishonest participant can use a vector of all 1s as its input and thereby learn the other participant's entire set! Hence, it should not be used in a standalone fashion by two mutually distrusting parties. Instead, such a protocol could be used as either a sub-protocol in a larger private computation where the participants do not control the inputs directly or do not see the outputs explicitly. Alternately, it could be combined with a *self-auditing step* (Section 4.8) to ensure that the result does not leak too much information or that neither input set is too large. One of the advantages of building our protocols using generic garbled-circuit techniques is that such extensions can easily be added.

4.5.3 Pairwise Comparisons

The running time of the BWA scheme scales linearly in the size of the universe (2^σ) over which the sets are defined. Thus, as the universe of elements grows, the BWA scheme becomes too inefficient to be useful. For large universes, we can use a Pairwise-Comparisons (PWC) protocol, shown in Algorithm 4. It performs comparisons between each pair of elements from the two parties' sets. The running time of PWC is quadratic in the set size (and linear in σ).

In Algorithm 4, the only part that needs to be implemented by a garbled circuit is the Equal function on line 6 which performs an equality test. An Equal circuit can be implemented by first XOR-ing the two σ-bit inputs to produce a σ-bit intermediate result. The negated-OR of these bits then indicates whether the two inputs match. Thus, an Equal circuit can be implemented using only $\sigma - 1$ non-free gates.

Algorithm 4 *PairwiseComparisons*(S, S′)

```
1:  for i ← 1 to S′.size do
2:      matched[i] ← False
3:  end for
4:
5:  for i ← 1 to S.size do
6:      for j ← 1 to S′.size do
7:          if ¬matched[j] and (Equal(S[i], S′[j])) then
8:              reveal(S[i])
9:              matched[j] ← True
10:             break
11:         end if
12:     end for
13: end for
```

To improve performance, our algorithm reveals each match as soon as it is found. This allows us to avoid performing further comparisons for any elements that have already been matched. (Recall that we assume each party's set contains no duplicates.) This optimization can potentially leak positional information about the elements in the parties' sets since the participants learn the order in which matching elements are found. To avoid this, each party randomly permutes its set before starting the protocol. Then, no information is revealed other than what could already be inferred from the result, namely, the elements in the intersection.

A drawback of the above "short-circuiting" optimization is that it substantially increases the round complexity since each **reveal** operation adds an extra round of communication. To benefit from this short-circuiting without the penalty of increased round complexity, we implement the protocol using two threads where the **reveal**s are done asynchronously while the main thread compares every possible pair of elements. Once a match is found by the **reveal** thread, the main thread is notified asynchronously to skip all unnecessary comparisons involving the matched element. Since the notification is asynchronous, it is possible that some Equal circuits are unnecessarily generated. However, our experiments show that the amount of wasted work is an insignificant fraction of the total work except for very small n.

Analysis. To understand the savings of the early reveal optimization, we provide a heuristic estimate for N_{Equal}, the expected number of calls to the Equal function. Let $\hat{n}$ be the size of the intersection. Since the two parties' sets S and S' are randomly shuffled before running Algorithm 4, the $\hat{n}$ elements in the intersection will, on average, be evenly distributed in S. Thus, we expect that on average the elements of S are ordered in such a way that there are $\hat{n}+1$ intervals of $(n-\hat{n})/(\hat{n}+1)$ non-matching elements each, with each interval separated by one of the matching elements. Assuming this to be the case, for each element in the i^{th} interval $(0 \le i \le \hat{n})$ the Equal function will be evaluated exactly $n-i$ times since it will be compared with all the $n-i$ currently-unmatched elements of S'. Each matching element in S is compared, on average, with half the remaining elements in S' before the match is found. Hence,

$$\begin{aligned} N_{\mathsf{Equal}} &\approx \sum_{i=0}^{\hat{n}} \frac{(n-\hat{n})(n-i)}{\hat{n}+1} + \sum_{i=0}^{\hat{n}-1} \frac{n-i}{2} \\ &= \frac{n-\hat{n}}{\hat{n}+1} \cdot \frac{(2n-\hat{n})(\hat{n}+1)}{2} + \frac{1}{2} \cdot \frac{(2n-\hat{n}+1)\hat{n}}{2} \\ &= \frac{(2n-\hat{n})^2+\hat{n}}{4}. \end{aligned}$$

Compared to a naïve implementation where all n^2 comparisons are performed, we see that short-circuiting saves roughly 75% of the comparisons if the two sets are identical, 45% if half the elements are identical, and 30% if $\frac{1}{3}$ of the elements are identical. Our experimental results (Figure 4.16) are consistent with this analysis.

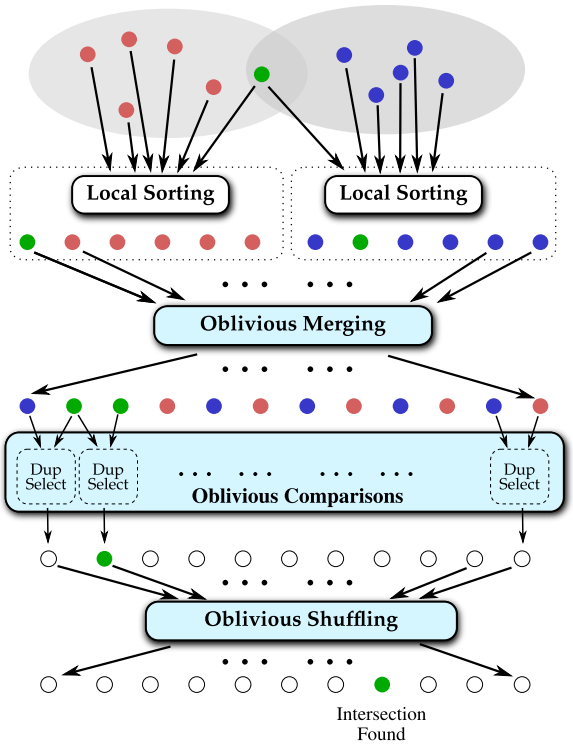


Figure 4.8: Sort-Compare-Shuffle Approach (parts requiring cryptographic computation are shaded).

4.5.4 Sort-Compare-Shuffle

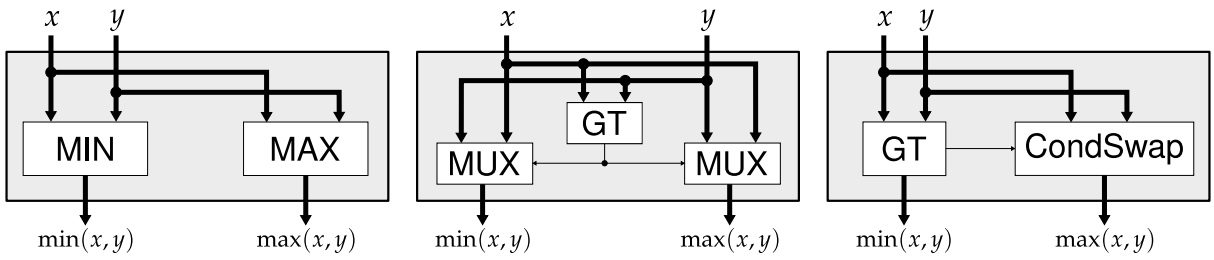
Although the pairwise-comparison protocol is intuitive and easy to implement, it requires $\Theta(n^2)$ comparisons and hence circuits with $\Theta(n^2)$ gates. Here we present PSI protocols that require only $\Theta(n \log n)$ element comparisons. These protocols take advantage of the observation that each participant can locally sort their own input set. We use this extra information to improve efficiency by breaking the task into three sequential sub-tasks as shown in Figure 4.8.

In each of the sort-compare-shuffle protocols of this section, each party begins by locally sorting their set. The parties then implement an oblivious merging network to sort the union of their sets, taking advantage of the fact that both input sets are sorted. Next, we use garbled circuits to compare neighboring elements in the sorted sequence to find all the matches. Directly outputting the matches at this stage would, however, reveal information about elements that are *not* in the intersection. (For example, if the parties learn that the first two elements in the sorted list match, this would reveal to the first party that the second party's set does not contain any elements smaller than the first matched element.) Thus, we obliviously shuffle the list of matched elements so that the positions of the matched elements are not revealed.

4.5.4.1 Sorting

The challenge of doing oblivious sorting using a garbled-circuit approach is that the sorting must be done by a sorting algorithm that uses a *fixed* (i.e., oblivious) sequence of comparisons. Most commonly used sorting algorithms do not lead to a size-optimal circuit. However, sorting networks [6] provide a fast circuit implementation of sorting. We further take advantage of the property that each party's inputs are independently sorted in designing a circuit that merges the two sorted lists to produce the full sorted list.

The basic module of a sorting network is a 2-Sorter, which sorts two σ-bit inputs. Figure 4.9(a) depicts a straightforward implementation of a 2-Sorter circuit. This design uses 4σ non-free binary gates to sort two σ-bit numbers, since the MIN and MAX circuits each use 2σ non-free gates [63].



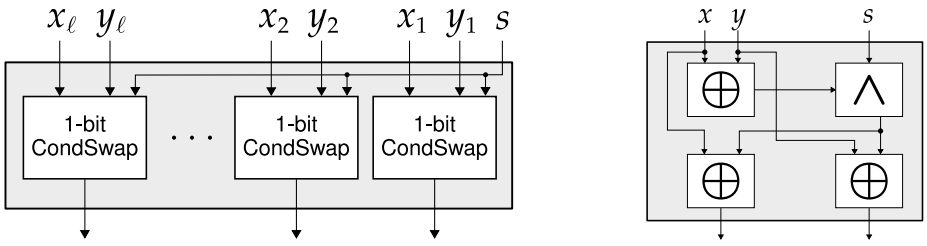
(a) Naïve (b) Better (c) Best

Figure 4.9: The design of a 2Sorter.

Because the MIN and MAX circuits each contain a GT (greater than) circuit, and share the same input, so we can eliminate one GT component to reduce the cost to 3σ non-free binary gates as shown in Figure 4.9(b). Furthermore, the two MUXs are unnecessary since their outputs are correlated. Based on this insight, we arrive at the final 2-Sorter design shown in Figure 4.9(c). It uses a conditional-swap circuit CondSwap (Figure 4.10), where a CondSwap circuit with σ-bit output (Figure 4.10(a)) is composed of σ parallel CondSwaps with 1-bit output (Figure 4.10(b)). The latter requires only one non-free gate. Thus, the overall cost of the 2-Sorter circuit is reduced to 2σ non-free binary gates. Kolesnikov and Schneider [64, 65] also designed a conditional-swap circuit (see [64, Fig. 2(b)]). Our CondSwap circuit has an explicit selection input bit, whereas in their case the selection bit is hardwired by the circuit generator.

Since the two input sequences provided by the parties are pre-sorted, we can sort their union using a *bitonic merger* [6] rather than having to use a full-fledged sorting network. A sequence is said to be *bitonic* if there is at most one extremum element and the two subsequences divided by this extremum element increase monotonically. As a specific example, the sequence that results from concatenating a sequence sorted in increasing order with a sequence sorted in decreasing order is bitonic.

Figure 4.11 depicts how a bitonic sequence of eight numbers is sorted by a bitonic merger. A bitonic merger for $2n$ inputs uses exactly $n\log(2n)$ 2-Sorter circuits (assuming n is a power of 2). Thus, we can



(a) σ-bit CondSwap (b) 1-bit CondSwap

Figure 4.10: CondSwap Circuits.

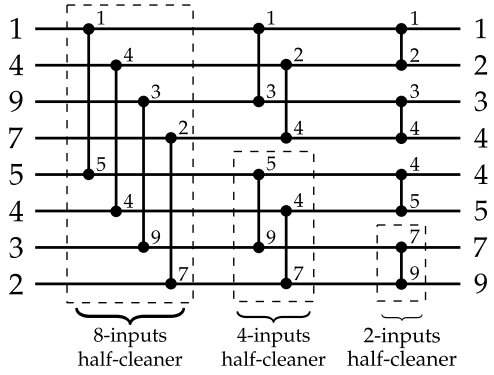


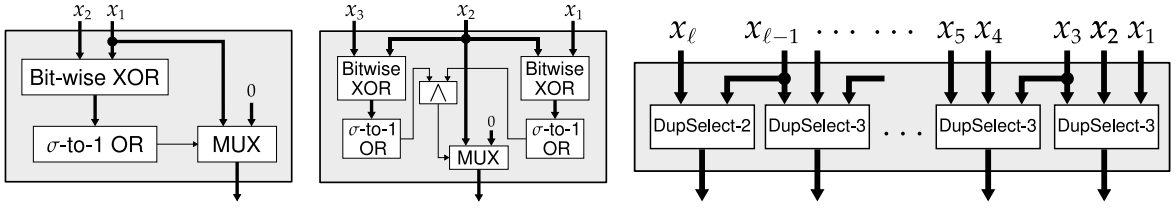
Figure 4.11: Example of merging a bitonic sequence.

construct a circuit that merges two lists of n sorted σ-bit elements into a sorted list of $2n$ elements using $2\sigma n \log(2n)$ non-free binary gates.

4.5.4.2 Filtering Matching Elements

After all $2n$ elements are in sorted order, we know that any elements in the intersection must be adjacent. Thus, to find the intersection we can use a duplicate-selection circuit (DupSelect-2) that takes as input two elements, $x_1, x_2 \in \{0,1\}^\sigma$, and outputs x_1 $(= x_2)$ if they are equal and 0^σ otherwise. (This assumes that 0^σ is not a valid element in the input set. If necessary, we can increase σ by one and remap elements to ensure this.)

Figure 4.12(a) shows the design of a DupSelect-2 circuit. Since we have $2n$ elements as input to this stage, $2n-1$ DupSelect-2 circuits are needed to identify all items in the intersection. As each DupSelect-2 circuit requires $2\sigma - 1$ non-free binary gates, the total cost of this phase as described is $(2\sigma-1)(2n-1)$.



(a) DupSelect-2 (b) DupSelect-3 (c) Comparison-based Filtering

Figure 4.12: Design and use of DupSelect-2 and DupSelect-3 circuits.

We next show how to reduce this cost by taking advantage of the property that the initial input sets have no repeated elements. This implies that for every three consecutive elements in the sorted sequence, there can be at most one match. To take advantage of this, we define a 3-input version of the duplicate-selection circuit, called DupSelect-3, as follows:

$$\text{DupSelect-3}\ (a,b,c) = \begin{cases} b & \text{if } a = b \text{ or } b = c \\ 0^\sigma & \text{otherwise} \end{cases}.$$

Figure 4.12(b) shows the design of a DupSelect-3 circuit using $3\sigma - 1$ non-free binary gates. Since we start with $2n$ elements as input to this stage, we need $n-1$ DupSelect-3 circuits and one DupSelect-2 circuit to identify all the elements in the intersection (see Figure 4.12(c)). This reduces the total number of non-free gates needed for this phase to $(3n-1)\sigma - n$. Another benefit of this design is that it produces only n output elements, rather than the $2n-1$ output elements that would be produced using the DupSelect-2 design. This reduces the size of the circuit needed in the subsequent oblivious shuffling phase (see below) by about 50%.

It is natural to ask whether it is possible to save even more gates by defining "higher-order" DupSelect circuits. We investigated it but found that this is not the case. The reason is that we save gates by cutting a MUX when we go from 2 to 3 inputs by exploiting the fact that within every three consecutive numbers there can be at most one match; this is no longer true once we look at four consecutive numbers. In fact, since there may be up to n items in the intersection, the number of outputs of this stage must clearly be at least n. Therefore, it does not help to combine more duplicate-selection circuits.

4.5.4.3 Shuffling

Following the filtering phase, we (implicitly) have a list of n elements that contains all $\hat{n}$ elements in the intersection, in sorted order, interleaved with an additional $n - \hat{n}$ occurrences of 0^σ. This list of elements cannot yet be revealed to the parties, however, since the positions of the 0-elements and the elements in the intersection may leak information about the parties' initial sets: for example, if the first element in the list is some match $x_1 \neq 0^\sigma$, this reveals that x_1 was the minimum element in both parties' sets. It is therefore necessary to destroy positional information before the elements are revealed.

We explore two general strategies for doing this: sorting the n intermediate values (Section 4.5.4.4), or randomly permuting them. For implementing the random permutation, we analyze strategies based on homomorphic encryption (Section 4.5.4.5) and using an oblivious shuffling network (Section 4.5.4.6).

4.5.4.4 Sorting

One way to hide positional information is to use an oblivious sorting network to sort the output sequence of the filtering phase (with 0^σ taken, say, to be minimal). This guarantees that no positional information leaks, since the sorted output could be generated from the intersection itself. Jónsson et al. [62] also use this general strategy in their work. As we show in Section 4.5.4.6, our circuit-based shuffling scheme is substantially more efficient than sorting-based approaches.

Batcher's sorting network provides a way to sort using $\Theta(n \log^2 n)$ gates [6]. Another possibility is to use the randomized Shellsort algorithm of Goodrich [43], which uses $\Theta(n \log n)$ gates but has non-zero error probability (corresponding to a small leak of information). We explored both these possibilities, but found that they are less efficient than the shuffling network presented in Section 4.5.4.6. In principle, sorting can also be done with $\Theta(n \log n)$ gates using the AKS sorting network [1], but the huge constant factor makes this approach impractical.

One scenario where sorting could be preferable, however, is when the size $\hat{n}$ of the intersection is small relative to the size n of the input sets. In that case sorting can be done using $n/\hat{n}$ calls to a $2\hat{n}$-sorter (that sorts $2\hat{n}$ elements), with total gate count (assuming Batcher's network is used for the $2\hat{n}$-sorter) of $n \log^2(2\hat{n})$. Though generally we cannot assume that $\hat{n}$ is small, it would be inexpensive to compute $\hat{n}$ securely (using a garbled circuit) after the filtering phase, at which point the parties could decide whether to use a sorting-based approach or a shuffling approach for the final phase. We do not explore this further.

4.5.4.5 Homomorphic Shuffling

Sorting actually does more work than necessary, since it is only necessary to hide positional information about the matches. We can do better by randomly permuting the elements rather than sorting them. In this and the next section, we consider two approaches to obliviously shuffle the results.

Our first shuffling approach uses homomorphic encryption to achieve linear asymptotic complexity. We begin by dividing each output from the end of the filtering phase into two secret shares, with one share given to each party. This can be done within a garbled-circuit computation as follows: Denote the intermediate results at the end of the filtering phase as $m_1, \ldots, m_n$, and recall that at this point neither party knows these values since they are encoded as part of the garbled-circuit computation. One party will provide an additional n random values $r_1, \ldots, r_n$ as input (at the beginning of the garbling stage). The garbled circuit is then extended so as to compute $r'_i = m_i + r_i$, with the other party learning r'_i. Note that r_i, r'_i form two shares of m_i. To ensure security, r_i must be sampled from a sufficiently large domain. Choosing r_i as a random $(\sigma + k)$-bit integer suffices to give statistical security $O(n \cdot 2^{-k})$.

Input to Alice: $r_1, r_2, \cdots, r_n$.
Input to Bob: $r'_1, r'_2, \cdots, r'_n$.
(for all i, we have $r_i + m_i = r'_i$, where m_i is the i-th number output by the filtering phase)

Output of Alice**:** The m_i's in random permuted order.
Output of Bob**:** $\perp$ (or, if desired, the m_i's in sorted order).

Preparation:
Alice chooses a key pair $\langle \mathsf{pk}_{P_1}, \mathsf{sk}_{P_1} \rangle$ and sends pk_{P_1} to Bob.

Execution:

1. Alice encrypts the r_i's $(1 \leq i \leq n)$ with her public key and sends the $[\![r_i]\!]$'s to Bob.
2. Bob computes $[\![m_i]\!] = [\![r'_i - r_i]\!] = [\![r'_i]\!] \cdot [\![r_i]\!]^{-1}$.
3. Bob randomly permutes the $[\![m_i]\!]$'s $(1 \leq i \leq n)$, and sends the resulting shuffled ciphertexts back to Alice.
4. Alice receives and decrypts the ciphertexts to output the m_i's $(1 \leq i \leq n)$.
5. (If desired) Alice sorts the m_i's and sends the result back to Bob.

Figure 4.13: Homomorphic-encryption-based shuffling protocol.

Now one party holds $r_1, \ldots, r_n$ and the other holds $r'_1, \ldots, r'_n$, with $m_i = r'_i - r + i$ for all i. The parties then execute the homomorphic-encryption-based shuffling protocol described in Figure 4.13. Throughout this protocol only one party's (e.g., Alice's) public key is required, so for simplicity we use $[\![x]\!]$ to denote $[\![x]\!]_{\mathsf{pk}_{P_1}}$, the encryption of x using Alice's public key pk_{P_1}. The key idea of this shuffling protocol is that the shuffler (Bob) cannot decrypt the ciphertexts he shuffles, whereas Alice (who knows the private key) does not know how the other party shuffled the ciphertexts.

Say the r_i's are λ-bit integers. Since each λ-bit full-adder used to perform the additive sharing requires λ binary AND gates, the secret-sharing phase altogether requires λn non-free binary gates. The homomorphic-encryption protocol in Figure 4.13 uses two rounds of communication, each round of which communicates n ciphertexts, and uses $O(n)$ public-key operations. Although this approach has asymptotic complexity linear in n, the actual cost of the best known homomorphic encryption schemes remains very high (see Section 4.5.5), and for the parameters we consider this protocol performs worse than the pure garbled-circuit protocol described in the next section which uses $\Theta(n \log n)$ symmetric-key operations. The other drawback with this approach is that it requires homomorphic encryption which abandons our goal of using only generic secure computation to enable easy integration with other secure computations.

4.5.4.6 Shuffling Network

Here we explore an alternate approach to random shuffling that uses $\Theta(n \log n)$ symmetric-key operations and remains a pure garbled-circuit protocol. The basic idea is to implement an oblivious random shuffling of the elements using a *switching network*. A switching network can be viewed as a fixed circuit that takes n inputs

along with an additional set of "control bits," each of which determines whether some fixed pair of elements is swapped or not. By setting the control bits appropriately, any desired permutation on the n inputs can be realized. In our setting the n inputs will be the n sorted elements from the end of the filtering stage, and one of the parties will choose a random permutation and then set the control bits so as to realize this permutation. The second party will receive as output the n elements, permuted according to the chosen permutation. If the first party should learn the output also, the second party applies another random permutation to the output elements (or simply sorts them) before sending them back. Switching networks can be constructed using $O(n \log n)$ gates [103].

The core component of a switching network is an oblivious swapper (2-Swapper) that takes as input two σ-bit values x and y, and an additional control bit s. If the value of s is 0, the output is x and y in their original order; if $s = 1$, the output is y and x in swapped order.

A switching network is simply a series of 2-Swappers (with independent control bits) applied to predetermined pairs of elements. A 2-Swapper circuit can be realized as a σ-bit CondSwap circuit (see Figure 4.10(b)). For our application, however, if we let the circuit generator set the control bits, then each AND gate in a CondSwap circuit can be replaced by the circuit generator with a 1-to-1 gate (which is either the identity or the 0-map, depending on the generator's secret s). Importantly, the type of the gate is known only to the circuit generator but is hidden from the circuit evaluator, so no information is leaked by this optimization. Combined with the garbled-row reduction (GRR) technique [86], the garbling of such a gate requires just a single ciphertext, which is one sixth of the cost of a 2-Sorter with GRR optimization. (Following the standard garbled-circuit approach, a unary gate would require two ciphertexts, but using the garbled-row reduction technique we can reduce this to a single ciphertext.)

The Waksman network [103], improving on the Beneš network [17], is a realization of a switching network using exactly $n \log n - n + 1$ 2-Swappers when n is a power of 2. (Constant-factor improvements when n is not a power of two were developed by Inria et al. [54], but we did not use those in our implementation.) Figure 4.14 illustrates a Waksman network for n inputs, assuming n is a power of 2. Its design is recursive: an n-input Waksman network is built out of two $\frac{n}{2}$-input Waksman networks (denoted by P_0 and P_1 in the figure) and $n - 1$ 2-Swappers (denoted by $\mathrm{I}_1, \cdots, \mathrm{I}_{\frac{n}{2}}$ and $\mathrm{O}_2, \cdots, \mathrm{O}_{\frac{n}{2}}$). Using the construction of a 2-Swapper circuit discussed earlier, the cost of the entire oblivious shuffling stage is only a small fraction (about 15%) of that spent in the oblivious sorting phase of the overall PSI protocol.

Note that choosing the control bits uniformly at random does not induce a random permutation. Instead, an algorithm is used to configure the control bits of a Waksman network to produce any of the $n!$ permutations of the n inputs. To induce a random permutation the circuit generator first chooses a random permutation π on n elements. It then uses the *ConfigureWaksman* function shown in Algorithm 5 to set the control bits,

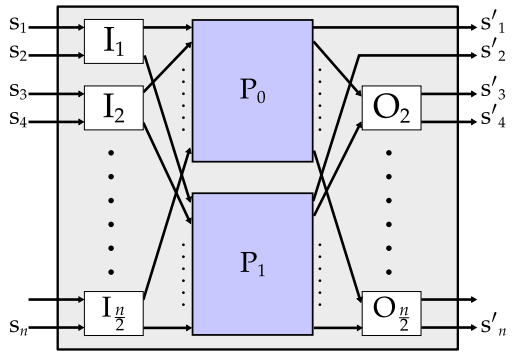


Figure 4.14: Waksman Network for n inputs.

Algorithm 5 *ConfigureWaksman*(n, π)

```
 1: init Boolean arrays I, O;
 2: π0 ← φ,    π1 ← φ;
 3: while ∃j such that Oj = ⊥ do
 4:    Oj ← 0; {Oj defaults to non-flip}
 5:    via ← 0;
 6:    while Ii/2 ≠ ⊥ do
 7:       [i, via] ← SetSwapper(I, j, via, π^-1);
 8:       π0 ← π0 ∪ {π^-1(j)/2 ↦ j/2};
 9:       [j, via] ← SetSwapper(O, i, via, π);
10:       π1 ← π1 ∪ {i/2 ↦ π(i)/2};
11:    end while
12: end while
13:
14: ConfigureWaksman(n/2, π0);
15: ConfigureWaksman(n/2, π1);
```

represented by the Boolean arrays I and O (corresponding to the 2-Swapper circuits in Figure 4.14). This algorithm sets the control bits in a recursive way. It starts from one of the unset swappers near an output port, say O_j, and sets O_j to *non-flip* position (line 4). Then, executing the inner *while* loop (lines 6–10), sets the configuration of $I_{\pi^{-1}(j)/2}$ by inspecting the parity of $\pi^{-1}(j)$. By looking at the permutation image of the other input to $I_{\pi^{-1}(j)/2}$, the algorithm can configure another swapper near the output ports. Therefore, the inner loop iterates over all swappers involved in a single sub-permutation, while the outer loop guarantees that all $n-1$ basic swappers pertaining to this level of the switch are traversed even if π consists of multiple sub-permutations.

The desired permutations of the component switches P_0 and P_1 are also recorded (lines 8, 10) as we set up the I, O swappers (line 7, 9). Thus, at the last two steps (lines 12–13), we only need to invoke *ConfigureWaksman* to deal with the internal swappers inside P_0 and P_1.

Algorithm 6 *SetSwapper*$(array, \iota, \varphi, \varpi)$

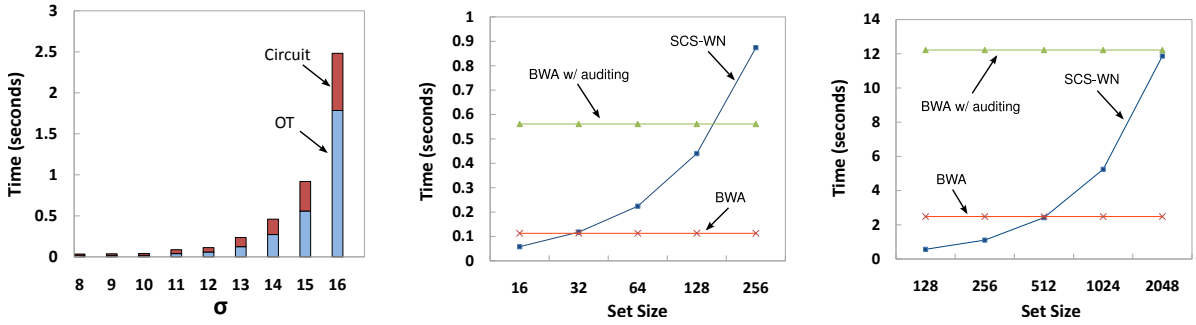
```
1: i ← ϖ(ι);
2: array_{i/2} ← (i%2) xnor φ;
3: return [i + ((i % 2 = 1) ? 1 : −1 ), 1 − φ];
```

The entire *ConfigureWaksman* algorithm is run locally by the circuit generator in our protocol, not within a garbled circuit. It involves no cryptographic operations, so the time it takes to execute is insignificant compared to the rest of the protocol. The configuration algorithms have negligible cost since they are executed as normal (unencrypted) computations and finish in linear time.

4.5.5 Experimental Results

We implemented each protocol and measured its performance on a range of inputs. In all experiments (except those where we fix the size of the intersection), both parties' sets consist of elements chosen at random (without replacement) from some fixed universe. All time measurements are the total time for the OT and garbled circuit execution, but do not include the one-time setup work for circuit object construction (about 1.2 seconds total for the most complex SCS-WN circuit) and OT extension protocol initialization (less than one second for the ultra-short security level). This time is not included in the results since (1) its cost does not depend on the size of the problem instance; and (2) it needs to be done only once for every client-server pair and circuit design.

Small Sets. We verified through experiments that the BWA protocol is indeed the best choice when the element space is small (up to about $\sigma = 20$). Figure 4.15 shows the running time of the BWA protocol for various sizes of σ, and compares its performance to that of the Sort-Compare-Shuffle scheme with Waksman-network shuffling, which we later show is the best protocol for larger element spaces. The BWA protocol is faster when the element space is limited but the set size is relatively large (e.g., $n > 40$ for $\sigma = 12$ and $n > 500$ for $\sigma = 16$).



(a) Performance of BWA (b) SCS-WN vs. BWA ($\sigma = 12$) (c) SCS-WN vs. BWA ($\sigma = 16$)

Figure 4.15: Set intersection for small element spaces.
The green lines depict the efficiency of secure auditing (Section 4.8).

Figure 4.16 compares the running time for the PWC and SCS-* protocols for $\sigma = 32$ and a range of small set sizes. The running time of the BWA protocol grows exponentially in σ and so other PSI protocols, including PWC, become more attractive as σ increases. Since the PWC scheme's performance also depends

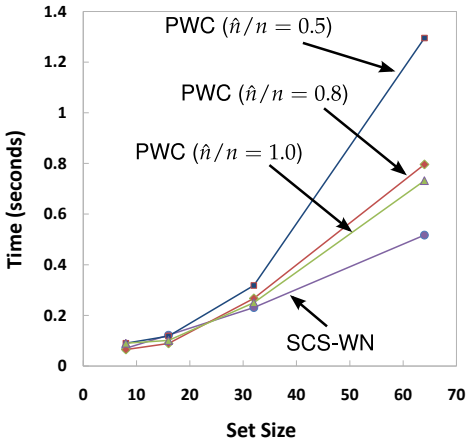
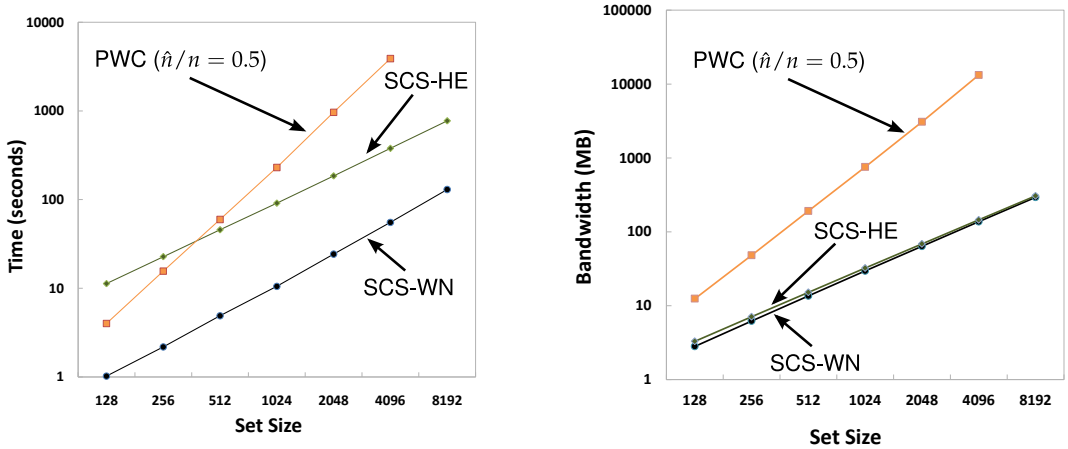


Figure 4.16: PSI — Small sets, $\sigma = 32$



(a) Running Time (b) Bandwidth

Figure 4.17: PSI — Large sets, $\sigma = 32$

on the size of the intersection, we include results for different ratios $\hat{n}/n$. In this figure, we only include the SCS-WN variant because it is the fastest of the SCS-* protocols.

Large Sets. Figure 4.17 shows the running time and bandwidth usage of different PSI protocols running on larger sets ranging from 128 to 8192 elements, with every set element represented by a 32-bit binary string. The only protocol whose expected running time depends on the elements in the parties' sets is the pairwise-comparison-based protocol where the performance improves with the size of the intersection. For this experiment, we fixed the $\hat{n}/n$-ratio to 0.5. Note that both axes are logarithmic scale.

The SCS-WN protocol for this range of parameters is superior to all other protocols by a significant advantage: over 7× faster than SCS with homomorphic-encryption-based shuffling (SCS-HE), and 10–70× faster than PWC. Contrary to expectations, the SCS-HE protocol does not save any bandwidth compared to SCS-WN which uses Waksman-network-based shuffling. In addition, we observe that because of our use of oblivious-transfer extension and our efficient OT implementation the OT step constitutes only about 5%

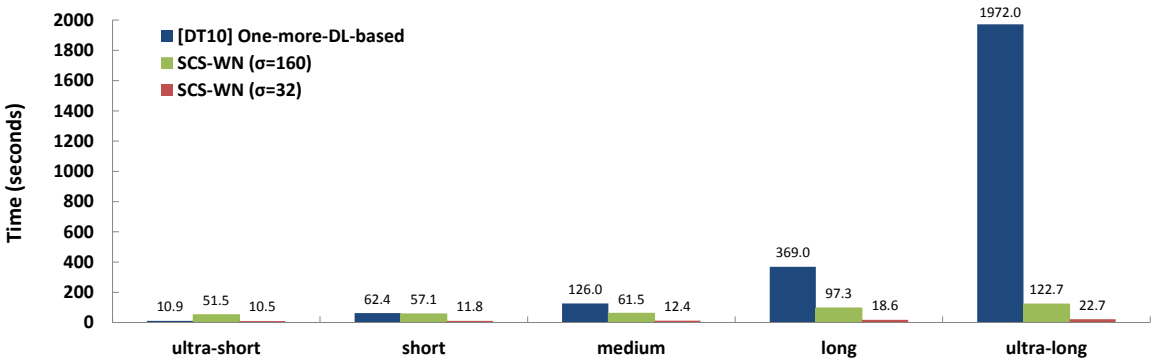


Figure 4.18: Comparision of SCS-WN and De Cristofaro-Tsudik protocol [32], $n = 1024$.
SHA-1 is used for ultra-short to medium term security. SHA-256 is used for long and ultra-long term security.

of the total cost. Using SCS-WN, we computed the intersection of two sets each containing more than one million 32-bit numbers ($n = 2^{20}, \sigma = 32$). This required executing a garbled circuit of 1.7 billion non-free gates, which completed in about 6 hours with each participant utilizing a single core of a typical desktop. This shows that our protocol makes large-scale privacy-preserving joint database search feasible for non-real-time applications with minimal hardware cost. When $\sigma = 160$ (effectively, $\sigma = \infty$ by first hashing elements to 160-bit strings using SHA-1), our results (see Figure 4.18) show that the time and bandwidth costs for SCS-WN and PWC will be about 5 times larger. Performance of the SCS-HE protocol, however, would be much less affected because the cost of HE-based shuffling, which dominates the cost of the protocol, is not affected by increasing σ from 32 to 160.

Since the timing results are sensitive to the implementations of particular cryptographic operations, we also calculate the numbers of expensive cryptographic operations required by each protocol. (Table 4.1 summarizes algebraically the number of gates needed for each of our protocols.) Table 4.2 summarizes the number of cryptographic operations required for each protocol. The small number of asymmetric operations used in garbled-circuit protocols is due to the operations for setting up the OT extension protocol, which only depend on the security parameters and can be precomputed offline once for each pair of protocol participants. The relatively high cost of asymmetric operations compared to symmetric encryptions means that even though SCS-WN requires approximately 2000 times the number of operations, the actual running time is lower or comparable for typical implementations.

[1]De Cristofaro and Tsudik [27] recently argued that our JVM based implementation could unfairly penalize the implementation of their large integer exponentiation based custom PSI protocol in these comparisons. They estimated that, for lower levels of security settings (key lengths ranging from 1024 to 3072 bits), the Java implementation of large integer modulo arithmetics could have inflicted about 5.5× performance penalty compared to the GMP C implementation. Note that the results reported in Table 4.2 are for our implementation of their protocol since they did not have an implementation available. But no data is reported on the penalty ratio of Java implementation of symmetric cryptographic operations over the C versions. We acknowledge that implementations of low-level cryptographic operations can have a large impact on performance results, but note that the important result here is that generic protocols can be competitive with the best known custom protocols.

	Exponentiations			Modular Inverses	Modular Mults	SHA
	short exponents	medium exponents	long exponents			
DT (one-more-DL-based)	–	5000	–	2049	4096	2048*
DT (one-more-RSA-based)	1024	–	2048	1024	1024	2048*
SCS-WN	–	$3k^{\dagger}$	–	$2k^{\dagger}$	$2k^{\dagger}$	18.34 M‡

Table 4.2: Number of expensive cryptographic operations, for $n = 1024$ and $\sigma = 160$.
[†] k is the security parameter used in OT extension, which ranges from 80 (for ultra-short) to 256 (for ultra-long).
[∗] The input messages are about p bits where p is the bit length of the asymmetric operations field size.
[‡] The input messages are about 2σ bits.

```
public static byte[] Cipher(byte[] key, byte[] msg) {
    byte[] state = AddRoundKey(key, msg, 0);
    for (int round = 1; round < Nr; round++) {
        state = SubBytes(state);
        state = ShiftRows(state);
        state = MixColumns(state);
        state = AddRoundKey(key, state, round);
    }

    state = SubBytes(state);
    state = ShiftRows(state);
    state = AddRoundKey(key, state, Nr);
    return state;
}
```

Figure 4.19: AES Cipher

4.6 Private AES

The background and potential applications of secure AES is introduced in Section 1.1.3. The high level operation of the cipher is shown in Listing 4.19 (based on [29]). It takes a 16-byte array `msg` and a large byte array `key`, which is the output of the AES key schedule. The variable `Nr` denotes the number of rounds (for AES-128, `Nr`=10).

4.6.1 Prior Work

Pinkas et al. [86] implement AES as an SFDL program, which is in turn compiled to a huge SHDL circuit consisting of more than 30,000 gates. Henecka et al. [47] used the same circuit, but obtained better online performance by moving more of the computation to the pre-computation phase. The best performance results they reported are 3.3 seconds in total and 0.4 seconds online per block-cipher evaluation.

4.6.2 Our Approach

Instead of constructing a huge circuit, we derive our privacy-preserving protocol implementation around the structure of a traditional program, following the code Listing 4.19. Our guiding principle is to identify the minimal subset of the computation that needs to be privacy-preserving, and only use expensive cooperative computation for those computations.

Overview. To make the implementation simpler, we explicitly group the wire labels of every 8-bit byte into *State*, representing the intermediate results of garbled circuits, so that they can be easily manipulated and passed around in the high level program. Compared to the original code (Listing 4.19), we only need to replace the built-in data type `byte` with our custom type `State` in building the code for implementing the garbled circuit. Since the state is represented by garbled wire labels, we can compose circuits implementing each execution phase to perform the secure computation.

The state of the `key`, the output of the key schedule, can be obliviously transferred from Alice to Bob so that the key scheduler can be executed by Alice alone. This enables us to replace the expensive privacy-preserving key schedule computation with less expensive oblivious transfers.

The `ShiftRows` subroutine can be safely executed independently by Alice and Bob on their own data. So nothing special is needed to make it privacy-preserving.

The `MixColumns` subroutine requires secure computation, but we design a circuit for this that uses only free XORs. The `AddRoundKey` subroutine is simply realized by a BitWiseXOR circuit which juxtaposes 128 binary free XOR gates.

SubBytes. The `SubBytes` component dominates the time for AES. To minimize the total execution time, I implemented `SubBytes` with an efficient circuit derived from the hardware circuit design of Wolkerstorfer et al. [105] (see that paper for details of the mathematical derivations behind this SBox implementation strategy). The two logical components of `SubBytes` are inverse over GF(2^8) and affine transformation over GF(2). The circuit we use to compute the inverse over GF(2^8) is given in Figure 4.20. In essence, GF(2^8) is viewed as an extension of GF(2^4), so that an element of GF(2^8) is mapped to its two GF(2^4) term representation, on which a series of operators including *inverse* over GF(2^4) are applied, and then mapped back to element in GF(2^8). In this circuit diagram, Map and Inverse Map circuits realize the bijections between GF(2^8) and $(\text{GF}(2^4))^2$; $\oplus$ and $\otimes$ mean *addition* and *multiplication* over GF(2^4), respectively. The affine transform over finite field GF(2) and all of the component circuits except for the $\otimes$ and GF(2^4)Inverse circuits can be implemented using free XOR gates alone. Since each $\otimes$ circuit has 16 non-free gates and each GF(2^4)Inverse 10, the total number of binary non-free gates per GF(2^8)Inverse circuit is $16 \times 3 + 10 = 58$. By adopting the optimized design given by Canright [23], the cost of this GF(2^8) inversion circuit can be further reduced to 36.

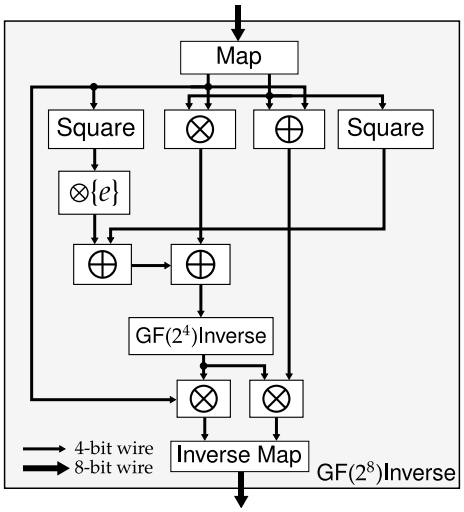


Figure 4.20: Inverse Circuit over GF(2^8).

MixColumns. The core functionality of `MixColumns` is $s'_c(x) = a(x) \otimes s_c(x)$, where $0 \leq c < 4$ specifies the column,

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\},$$

and $\otimes$ denotes multiplication over finite field GF(2^8). Let $s_c(x) = s_{3,c}x^3 + s_{2,c}x^2 + s_{1,c}x + s_{0,c}$ and $s'_c(x) = s'_{3,c}x^3 + s'_{2,c}x^2 + s'_{1,c}x + s'_{0,c}$. This is equivalent to

$$\begin{aligned}
s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\
s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}).
\end{aligned}$$

It follows that

$$\begin{aligned}
s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{02\} \cdot s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{02\} \cdot s_{2,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{02\} \cdot s_{3,c}) \oplus s_{3,c} \\
s'_{3,c} &= (\{02\} \cdot s_{0,c}) \oplus s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}).
\end{aligned}$$

The operation $\{02\} \cdot b$ where b is an arbitrary 8-bit number, known as *xtimes*, is defined as multiplying $\{02\}$ modulo $\{1b\}$ in GF(2^8). If $b = b_7 \cdots b_1 b_0$, and $z = z_7 \cdots z_1 z_0 = \{02\} \cdot b$, the output bits can be computed

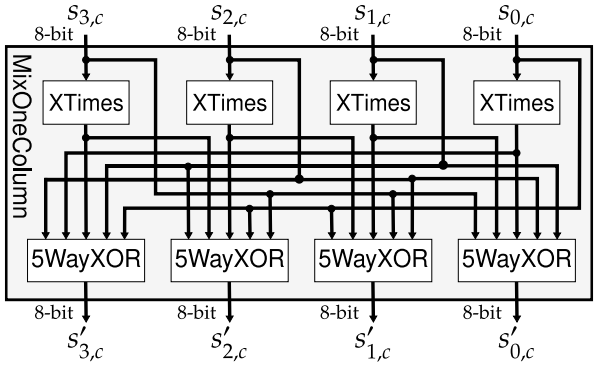


Figure 4.21: MixOneColumn Circuit.

using only free XOR gates:

$$z_7 = b_6, \quad z_6 = b_5, \quad z_5 = b_4, \quad z_4 = b_3 \oplus b_7,$$
$$z_3 = b_2 \oplus b_7, \quad z_2 = b_1, \quad z_1 = b_0 \oplus z_7, \quad z_0 = b_7$$

This can be computed using one three free XOR gates.

For every column of 4-byte numbers, the equations above are implemented by the MixOneColumn circuit (Figure 4.21). Each invocation of `MixColumns` involves processing four columns, so we can build the MixColumns circuit by juxtaposing four MixOneColumn circuits. Thus, the MixColumns circuit can be implemented using only free XOR gates.

4.6.3 Evaluation

With our design, the total number of binary non-free gates is $58 \times 16 \times 10 = 9280$ for both Alice and Bob. The overall time is 0.2 seconds (of which 0.08 seconds is spent on OT) without preprocessing, about 16 times faster than the best prior work.

4.7 Secure Minimum and Information Retrieval

Many biometric identification algorithms find the best match by computing a global minimum (or maximum) over a set of candidate difference (or similarity) scores. If the minimum (maximum) is upper-bounded (lower-bounded) by some given threshold, a legitimate match will be flagged. In addition, the match is usually followed by an information retrieval process to get the matching record. In the realm of secure biometric recognition, both steps need to be carried out without leaking any intermediate information. This motivates our secure minimum (Section 4.7.1) and backtracking (Section 4.7.2) protocols.

4.7.1 Secure Minimum

Secure minimum takes as input a vector of M integers, and outputs the largest one. A M-to-1 Min circuit is simply a tree of 2-to-1 Min circuits (with the latter being constructed as in [63]). However, in contrast to the work of Kolesnikov et al. [63], we only need to compute the minimum value rather than the *index* of the minimum value. This is a consequence of the backtracking protocol that we present in the next section. This allows us to reduce the number of gates by roughly $M - \log M$ overall.

Table 4.3 summarizes the number of non-XOR gates in each of our circuits.

2-to-1 Min	M-to-1 Min
$2k$	$2k(M-1)$

Table 4.3: The number of non-free binary gates in each circuit.

4.7.2 Backtracking

With a wire label denoting the match, the backtracking protocol allows to retrieve the information associated with the match obliviously without the overhead of propagating the indexes throughout the minimum circuit above.

In a conventional garbled circuit, wire signals (0 or 1) are denoted by randomly-chosen nonces known as *wire labels*. The bindings between wire labels and the wire signals are known by the circuit generator, but hidden from the circuit evaluator (except for the bindings for the final output wires which are disclosed to reveal the result). Wire labels are merely used for intermediate computation. However, these apparently meaningless nonces can be exploited in later stages of the protocol. We take advantage of the key property of garbled circuit evaluation: the evaluator only learns one of the two possible output wire labels for each gate, as determined by the obliviously-selected input wire labels and evaluation of the rest of the circuit. These wire labels can serve as keys for encrypting useful information.

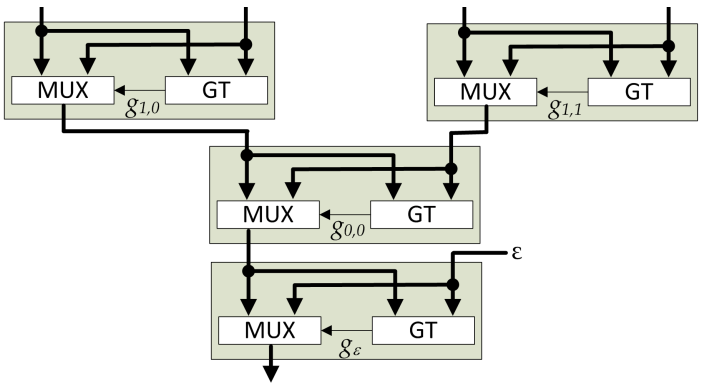


Figure 4.22: Example Find Closest Match Circuit

The wire labels of the output wires of GT comparison circuits in the n-to-1 Minimum tree can be used to reveal a path from the inputs to the minimum value. Figure 4.22 shows an example comparison tree for a four record database. In each of the 2-to-1 Min circuits the GT circuit takes two inputs and outputs a bit indicating which value is greater. We denote that bit as $g_{h,i}$ and the corresponding wire labels $\lambda^0_{h,i}$ (when the greater than comparison is false) and $\lambda^1_{h,i}$ (when the comparison is true). When the more closely matching entry match is on the left side of this gate, Bob learns $\lambda^0_{h,i}$; when it is on the right side he learns $\lambda^1_{h,i}$. The final gate in the diagram compares the best match with ϵ. We use the g_ϵ output to prevent Bob from learning any information from the backtracking tree when there is no match within the ϵ threshold.

Our backtracking tree protocol involves a tree generator (Alice), who produces and sends a tree encoding encrypted paths to the profile records, and a tree evaluator (Bob), who follows a single path through the tree to open the best matching profile record. To generate a backtracking tree, Alice starts by filling the leaf nodes (level 0) with the desired information corresponding to each database entry. Then, she fills in the internal nodes of a binary tree with those leaves, as illustrated by the left tree in Figure 4.23. Note that the structure of this tree is identical to that of the comparison tree in Figure 4.22.

Next, she generates new nonces for each non-leaf node in the tree, and encrypts those nonces with keys that combine the appropriate wire labels and the nonce of its parent node. The wire label used for node h, i (the i^{th} node at level h) is either $\lambda^0_{h,i}$ or $\lambda^1_{h,i}$, depending on whether it is the left or right child of its parent). Thus, the label pair Alice uses for each node comes from the labels of $g_{h,i}$ in the corresponding 2-to-1 Min circuit she generated for the match-finding protocol. The root node is encrypted using λ^0_ϵ, the label Bob will learn when the closest match is closer than ϵ. The right tree in Figure 4.23 shows the backtracking tree corresponding to the example circuit in Figure 4.22.

Starting from the root of the tree, which Bob can only open when $g_\epsilon = 0$, Bob can follow a single path through the tree, learning the keys along that path, and eventually the key needed to decrypt the encrypted record information at the leaf. When he evaluates the garbled SubReduceMin circuit, Bob obtains either $\lambda^0_{h,i}$ or $\lambda^1_{h,i}$ from each label pair. Since each key in the backtracking tree depends on a complete path from the root to that tree, this means Bob can only open a single path through the tree; specifically, the single path from the root to the leaf corresponding to the closest match.

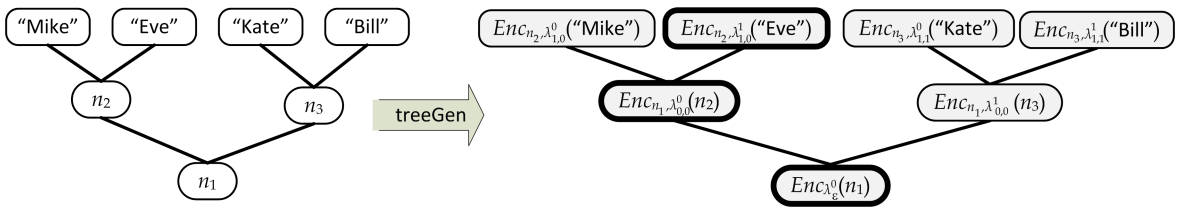


Figure 4.23: Backtracking Tree Example

Input to Tree-Generator: (1) an array $[\texttt{data}_1, \cdots, \texttt{data}_M]$ denoting M pieces of data stored in the database;
(2) a number of pairs $\left[\texttt{label\_pair}_1, \cdots, \texttt{label\_pair}_{M-1}\right]$, where $\texttt{label\_pair} \stackrel{\text{def}}{=} \langle \texttt{label}_0, \texttt{label}_1 \rangle$, organized as a tree.

Input to Tree-Evaluator: (1) a backtracking tree `tree`;
(2) a number of labels $[\texttt{label}_1, \cdots, \texttt{label}_{M-1}]$ organized as a tree.

Protocol Output: Tree-Evaluator learns $\texttt{data}_{i^*}$, where $d_{i^*} = \min_{i=1}^{M}(d_i)$. Tree-Generator learns nothing new.

Execution:

Tree-Generator		Tree-Evaluator
① `tree=treeGen(data, label_pair)`	② `tree` ⟶	③ $\texttt{data}_{i^*}$ = `treeEval(tree, label)`

Figure 4.24: The Backtracking Tree Protocol.

Figure 4.24 summarizes the backtracking protocol. The algorithms to generate and evaluate the tree are shown in Algorithm 7 and Algorithm 8. For the tree generation algorithm (Algorithm 7), the inputs a vector of the profile data to send and an array of the wire label pairs for each comparison gate in the generated circuit. The output is the backtracking tree, but only the node labels are transmitted to the evaluator. For the tree evaluation algorithm (Algorithm 8), the inputs are the tree (where the label for each node is the encrypted label in the generated tree) and the wire labels learned by the evaluator in evaluating the garbled circuit. The output is the decrypted profile information for the closest matching entry, if there is one within ϵ.

Security. The backtracking tree protocol is secure if both of the following properties hold:

1. The generator (Alice) gains nothing.
2. The evaluator (Bob) gains nothing other than the data associated with the closest match.

Algorithm 7 `treeGen(data, label_pairs)`

Require: `data.length` $= M$; `label_pairs.length` $= M - 1$; $M = 2^{\texttt{h}}$, where `h` is an integer.
1: Generate a perfect tree `tree` of size $2M - 1$.
2: Fill the M leaf nodes with the M values in `data`.
3: **for all** `node` in `tree` **do**
4: `node.nonce` $\stackrel{\text{U}}{\leftarrow} \{0,1\}^k$;
5: **end for**
6: **for** $\ell \leftarrow$ `h` $- 1$ to 1 **do**
7: **for all** `node` at level ℓ **do**
8: `lp` $\leftarrow$ `label_pairs[pos(node)]`, the labels for the gate corresponding to `node` in the `tree`;
9: `node.leftChild.label` $\leftarrow \texttt{Enc}_{\texttt{node.nonce}||\texttt{lp.label}_0}(\texttt{node.leftChild.nonce})$;
10: `node.rightChild.label` $\leftarrow \texttt{Enc}_{\texttt{node.nonce}||\texttt{lp.label}_1}(\texttt{node.rightChild.nonce})$;
11: **end for**
12: **end for**
13: **return** `tree`;

Algorithm 8 `treeEval(tree, wire_labels)`

```
1: msg ← 0;
2: current_node ← tree.root;
3: while current_node has children do
4:    m ← Dec_{msg||wire_labels[pos(current_node)]}(current_node.leftChild.label);
5:    if m is valid then
6:       msg ← m;
7:       current_node ← current_node.leftChild;
8:    else
9:       msg ← Dec_{msg||wire_labels[pos(current_node)]}(current_node.rightChild.label);
10:      current_node ← current_node.rightChild;
11:   end if
12: end while
13: return msg;
```

The first property trivially holds since Bob sends nothing back to Alice. The second property follows from two facts:

1. With a *semantically secure* encryption scheme, no information is leaked by the encryption (i. e., without also revealing the keys).
2. *Wire labels* in a garbled circuit convey no information unless their mappings to *wire signals* are known somehow. This follows from the garbled circuit security proof [69].

In every iteration of the loop in Algorithm 8, Bob only gets to know the nonce of one of current_node's two children, and proceeds using that value. The whole subtree of the failed branch remains unknown to Bob since the nonce is needed to open any configurations on that subtree. Thus, Bob can only follow a single path in the tree corresponding to the path leading to the closest match.

Note that if Alice's database is released in encrypted format beforehand as in the Improved Euclidean Distance protocol, Alice may not want Bob to learn extra information from the position of the matched records. Hence, Alice should randomly permute the order of database records before beginning the Euclidean distance protocol. This random permutation needs only to be done once, since once the database is permuted relative positions of opened records reveal no information.

4.7.3 Experimental Results

We set up the server and the client on separate machines connected by a LAN. Both machines are homogeneously configured, each with an Intel Xeon CPU (E5504) running at 2.0GHz. The JVMs are configured with a memory cap of 4GB, both on the server and the client. Our experiments are done on 16-bit integers with ϵ also set to a 16-bit integer.

Database Size (M)		**128**		**256**		**512**		**1024**	
Time/Bandwidth		s	KB	s	KB	s	KB	s	KB
Prep.	Min Circuit	0.35	None	1.09	None	2.95	None	6.28	None
	OT	0.52	21.91	0.48	21.91	0.48	21.91	0.45	21.91
Exec.	OT	0.13	237.13	0.25	467.69	0.61	928.82	1.38	1851.06
	Min Circuit	0.39	656.14	0.67	1313.64	1.58	2628.64	3.12	5258.63
	Backtracking	0.01	12.70	0.02	25.45	0.03	50.95	0.04	101.94

Table 4.4: Running Time (seconds) and Bandwidth (KB) for Minimum and Backtracking Protocols

Table 4.4 shows the running time and bandwidth usage for our secure minimum and backtracking protocol as a function of M, the number of input integers. As expected, the results in Table 4.4 confirm that the runtime cost scales approximately linearly with the size of the database.

4.8 Auditing

An advantage of using generic garbled circuits instead of a custom protocol is the relative ease with which the generic secure computation can be combined with subsequent computations in a privacy-preserving protocol. As an example, we describe in this section how simple auditing mechanisms can be incorporated into our PSI protocols.

The result of a private set intersection intrinsically leaks a great deal of information about the private input sets, especially when σ is small enough to allow easy probing. For small enough input universes, a dishonest participant can simply set its own input to be the set containing all values in the data universe and learn the other participant's entire set. To mitigate excessive information leaks, auditing logic could be incorporated into any of the pure garbled-circuit protocols we have described.

A simple auditing policy would place a threshold on the maximum size $\hat{n}$ of the intersection that would be revealed. If the size of the result exceeds this threshold, then no output is revealed (and so the participants would only learn that the size of the intersection exceeds the allowed threshold). Such self-auditing logic would be very cheap to implement with garbled circuits, but appears to be difficult to incorporate into custom-designed PSI protocols.

As an initial study, we developed prototypes realizing the threshold-based auditing scheme just discussed. The extra work here is to obliviously calculate $\hat{n}$ (main cost) and then compare this value to a threshold using a comparison circuit (insignificant cost). For the BWA scheme, $\hat{n}$ is calculated by a Counter circuit that sums up the output bits of all the AND gates. For the SCS-* family of schemes described in Section 4.5.4, the input signals to the MUX circuits inside the duplicate-selection circuits (cf. Figures 4.12(a) and 4.12(b)) are summed using a Counter circuit. As an optimization, our Counter circuit lazily increases the number of bits used to represent its internal state. Constructing it in this way, the Counter circuit uses $n \log n - n$ non-free

gates. Note that when the thresholds are known for specific applications, the cost of the Counter circuit can be cut further since there is no reason to represent results that exceed the threshold.

Our experiments show that our size-based auditing circuits incur no measurable performance overhead for the SCS-* protocols. For the BWA scheme, the cost of the auditing is significant because the underlying BWA protocol is so fast. For $12 \leq \sigma \leq 16$, adding size-based auditing increases the overall time by roughly a factor of 5 (e.g., for $\sigma = 16$, it takes 2.48 seconds to run the BWA protocol without auditing but 12.22 seconds with auditing). This is consistent with our analysis: BWA with auditing uses a total of $\sigma 2^\sigma$ non-free gates compared to 2^σ non-free gates without auditing, but the garbled-circuit portion of the basic BWA protocol constitutes only 35–45% of the total running time.

4.9 Summary

We have applied the general optimization techniques presented in Chapter 3 to solve a number of field problems. Experiments show that the techniques brings about significant performance improvements (Figure 4.25). The rough measurement of the speed of garbled circuits across different applications is about 10μs per garbled gate. In addition, we have tested it scales to more than 10^9 gates. Secure computation is still expensive compared to standard computation; but, with the techniques developed here, many applications are now within reach even on commodity computers.

Problem	Best Previous Result	Our Result	Speedup
Hamming Distance (Face Recognition, Genetic Dating) – two 900-bit vectors	213s [SCiFI, 2010]	**0.051s**	**4176x**
Levenshtein Distance (genome, text comparison) – two 200-character inputs	534s [Jha+, 2008]	**18.4s**	**29x**
Smith-Waterman (genome alignment) – two 60-nucleotide sequences	[Not correctly implemented]	**447s**	**-**
AES Encryption	3.3s [Henecka+, 2010]	**0.2s**	**16.5x**
Private Set Intersection (medium level of security)	126s [De Cristofaro+, 2010]	**12.4**	**10x**

Figure 4.25: Summary of results in the semi-honest threat model.

Chapter 5

Strengthening the Threat Model

The results from the previous chapter all assume the semi-honest adversary model. This enables efficient solutions, but is too weak an adversary model to be appropriate for most realistic scenarios. Existing protocols secure in presence of active adversaries exhibit a slowdown of several orders of magnitude compared to protocols with semi-honest security. The slowdown is not only due to increased computation and communication; a (potentially) more significant issue is the memory usage required by some of the protocols. As an example, the Lindell-Pinkas protocol [68] requires hundreds of copies of the garbled circuits to be transmitted before verification and evaluation, and so does not appear to be compatible with pipelined circuit execution (a technique that makes secure computation memory efficient [50]), at least not without introducing additional overhead.

In this chapter, we discuss a threat model that offers security guarantees lying between the existing semi-honest and fully malicious models. It provides meaningful (but weaker than full) security guarantees in the presence of malicious active adversaries, but performance close to what can be achieved with semi-honest protocols. It has orders of magnitude better performance than the best malicious model protocols.

5.1 Related Work

Most implementations of generic secure two-party computation have targeted the semi-honest threat model [74, 47, 50], and have used protocols based on Yao's garbled-circuit approach. As is described in the previous two chapters, the scalability and efficiency of garbled-circuit protocols can be improved to the point where large applications can be run on commodity desktops.

[0]This chapter is based on our paper "Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution" [51].

Several approaches have been proposed for achieving security against malicious adversaries [41, 59, 68, 81, 94, 70, 67, 82], some of which have been implemented [71, 86, 94, 82].

Several directions have been explored to achieve better trade-offs between security and efficiency. The *covert* model (Section 2.1.3) is among the first few attempts in the direction. More closely to our work, Mohassel and Franklin [76] proposed a relaxed definition of security in which, informally, a malicious adversary may be able to learn a small number of additional bits of information about the honest party's input, beyond what is implied by the output. This definition may suffice for many realistic scenarios in which secure computation would be used. Note that even fully secure protocols leak information about the honest party's input in the function output. Depending on the specific function and the control the adversary has over its own input this information leakage may be substantial and hard to characterize.

5.2 Dual-Execution Protocols

Informally, a secure-computation protocol needs to satisfy two properties: *privacy*, which ensures private inputs are not revealed improperly, and *correctness*, which guarantees the integrity of the final output. Yao's (semi-honest) garbled-circuit protocol is easily seen to provide *one-sided privacy* against a malicious circuit generator as long as an OT protocol secure against malicious adversaries is used, and the evaluator does not reveal the final output to the circuit generator. Note that this usage of garbled circuit provides privacy, but does not provide any correctness guarantees. A malicious generator could construct a circuit that produces an incorrect result without detection. Hence, this approach is insufficient for scenarios where the circuit generator may be motivated to trick the evaluator by producing an incorrect result. It thus remains to provide correctness guarantees for an honest circuit evaluator against a possibly malicious generator, and to provide a way for both parties to receive the output while continuing to provide privacy guarantees against a malicious generator.

Mohassel and Franklin's *dual-execution* (DualEx) protocol [76] provides a mechanism to achieve these guarantees. The protocol involves two independent executions of the semi-honest garbled-circuit protocol, where each participant plays the role of circuit generator in one of the executions. The outputs obtained from the two executions are then compared to verify they are identical; if so, each party simply outputs the value it received. Intuitively, this may leak an extra bit of information to an adversary who runs the comparison protocol using an incorrect input.

Mohassel and Franklin left some details of the protocol unspecified, and did not give a proof of security. They also did not provide any implementation of their approach. Next, we describe the protocol and method for comparing outputs in more detail.

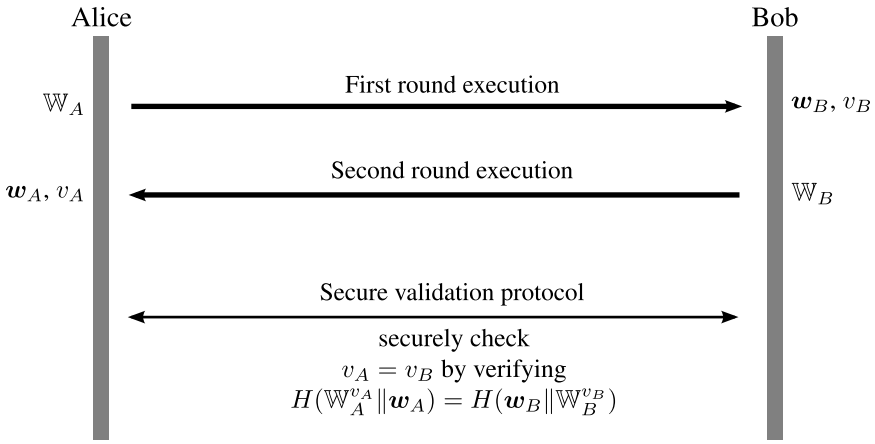


Figure 5.1: DualEx protocol overview (informal).

5.2.1 Notation

We write a set of wire-label pairs as a matrix:

$$\mathbb{W} = \begin{pmatrix} w_1^0 & w_2^0 & \cdots & w_\ell^0 \\ w_1^1 & w_2^1 & \cdots & w_\ell^1 \end{pmatrix}.$$

A vector of wire labels is denoted as

$$\boldsymbol{w} = (w_1, w_2, \ldots, w_\ell)\,.$$

If $v \in \{0,1\}^\ell$ is a string and $\mathbb{W}$ is a matrix as above, then we let

$$\mathbb{W}^v = (w_1^{v_1}, \ldots, w_\ell^{v_\ell})$$

be the corresponding vector of wire labels.

5.2.2 Protocol

Assume the parties wish to compute some function f, and (for simplicity) that each party holds an n-bit input and that f produces an ℓ-bit output. Figure 5.1 depicts an overview of the basic DualEx protocol, which was proposed by Mohassel and Franklin [76, Section 4.1]. The protocol consists of two separate runs of a particular semi-honest protocol plus a final stage for verifying that certain values computed during the course of the two semi-honest executions are identical.

A more detailed description of the DualEx protocol is shown in Figure 5.2. The protocol is conceptually divided into three stages: the first run, the second run, and the secure validation. For the sake of performance, however, our implementation executes the first two stages concurrently, using pipelining to overlap the

Input to Alice**:** the private input x.
Input to Bob**:** the private input y.

Output to both Alice **and** Bob**:** $f(x, y)$, or $\perp$ if cheating is detected.

Execution:

1. Alice and Bob run the semi-honest garbled-circuit protocol (Figure 5.3) where Alice plays the role of circuit generator (P_1), and Bob plays the role of circuit evaluator (P_2). Alice knows the 2ℓ output-wire labels she generated, $\mathbb{W}_A$, while Bob learns ℓ output-wire labels $\boldsymbol{w}_B$ and an output $v_B \in \{0,1\}^\ell$. (If both parties are honest, $\boldsymbol{w}_B = \mathbb{W}_A^{v_B}$.)
2. Alice and Bob invoke the semi-honest garbled circuit protocol again, swapping roles. Alice learns the output v_A along with labels $\boldsymbol{w}_A$, while Bob knows the label pairs $\mathbb{W}_B$. (If both parties are honest, then $\boldsymbol{w}_A = \mathbb{W}_B^{v_A}$, and also $v_A = v_B$.)
3. Alice and Bob run a "validation protocol" (i.e., an equality test), secure against malicious adversaries. (See Figures 5.4 and 5.5 for one possible instantiation.) Alice uses input $\mathbb{W}_A^{v_A} \| \boldsymbol{w}_A$ and Bob uses input $\boldsymbol{w}_B \| \mathbb{W}_B^{v_B}$. If the protocol outputs `true`, then Alice outputs v_A and Bob outputs v_B. Otherwise, the honest party has detected malicious behavior and outputs $\perp$.

Figure 5.2: DualEx protocol

circuit-generation and circuit-evaluation work for each party. (As long as the oblivious transfers are done sequentially, our security proof is unaffected by performing the two garbled-circuit executions in parallel. The reason is that our security proof holds even against a rushing adversary who waits to receive the entire garbled circuit from the honest party before sending any of its own garbled gates.) We stress that the parties run each of the first two stages to completion — even if an error is encountered — so that no information is leaked about the presence or absence of errors in an execution. If an error is detected that prevents normal progress, the execution continues using random values.

The DualEx protocol uses a specific garbled-circuit protocol with an oblivious-transfer sub-protocol secure against malicious adversaries (see Figure 5.3). After an execution of this protocol, only P_2 learns the output (but is uncertain about its correctness), while P_1 learns nothing. In this version, the result $f(x, y)$ is revealed to P_2 (Bob in the first execution as defined in Figure 5.2) even if cheating by P_2 is detected during the equality-checking protocol. This does not violate our definition of security, however for many scenarios this property would be undesirable. Section 5.4 presents some heuristic enhancements to the basic protocol that address this issue by limiting the amount of information either party can obtain during the protocol execution.

5.2.3 Secure Output Validation

The goal of the secure validation protocol is to verify the correctness of the outputs Alice and Bob obtained in the previous stages. The validation protocol consists of an equality test between certain *output-wire labels* held by each of the parties. Since half the output-wire labels chosen by the garbled-circuit generator are never learned by the circuit evaluator (and since output-wire labels are chosen at random) this has the effect

Input from P_1**:** private input x.
Input from P_2**:** private input y.

Output to P_1**:** the output wire key pairs $\mathbb{W}_A = \begin{pmatrix} w^0_{A1} & w^0_{A2} & \cdots & w^0_{A\ell} \\ w^1_{A1} & w^1_{A2} & \cdots & w^1_{A\ell} \end{pmatrix}$.

Output to P_2**:** $v_B \in \{0,1\}^\ell$ representing the value of $f(x,y)$, and output-wire labels $\boldsymbol{w}_B = (w^{v_{B1}}_{A1}, w^{v_{B2}}_{A2}, \cdots, w^{v_{B\ell}}_{A\ell})$.

Execution:

1. P_1 and P_2 run a garbled-circuit protocol where P_1 plays the circuit *generator*'s role and P_2 the circuit *evaluator*'s role.
2. P_1 and P_2 execute a malicious OT protocol (with P_1 the *sender* and P_2 the receiver) to enable P_2 to learn the wire labels corresponding to P_2's input y. Then P_2 evaluates the garbled circuit to learn output-wire labels $\boldsymbol{w}_B$.
3. P_1 computes
$$\begin{pmatrix} H(w^0_{A1}) \cdots H(w^0_{A\ell}) \\ H(w^1_{A1}) \cdots H(w^1_{A\ell}) \end{pmatrix}$$
for H a random oracle, and sends it to P_2 so that it can use $\boldsymbol{w}_A$ to learn v_B.
4. If P_2 detects cheating at any point during the protocol, he does not complain but instead just outputs completely random v_B and $\boldsymbol{w}_B$.

Figure 5.3: Semi-honest garbled-circuit sub-protocol

Input to Alice**:** $\mathbb{W}^{v_A}_A, \boldsymbol{w}_A$.
Input to Bob**:** $\boldsymbol{w}_B, \mathbb{W}^{v_B}_B$.

Output to both Alice **and** Bob**:**

$$\begin{cases} \mathsf{true}, & \text{if } \mathbb{W}^{v_A}_A = \boldsymbol{w}_B \textbf{ and } \boldsymbol{w}_A = \mathbb{W}^{v_B}_B; \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

Execution:

1. Alice computes $h_1 = H(\mathbb{W}^{v_A}_A \| \boldsymbol{w}_A)$;
2. Bob computes $h_2 = H(\boldsymbol{w}_B \| \mathbb{W}^{v_B}_B)$;
3. Alice and Bob uses an equality test (secure against malicious adversaries) to compare h_1 and h_2. If they are equal, Alice and Bob both output true; otherwise they output false.

Figure 5.4: An instantiation of the secure-validation protocol.

of preventing an adversary from (usefully) changing their input to the equality test. In an honest execution, on the other hand, since both executions of the garbled-circuit sub-protocol are computing the same function on the same inputs, the inputs to the equality test will be equal.

The equality test will be done by first (*a*) computing a hash of the inputs at both sides, and then (*b*) comparing the hashes using an equality test that is secure against malicious adversaries. (See Figure 5.4.) If the hash used in the first step is modeled as a random oracle (with sufficiently large output length), then it is possible to show that this results in an equality test for the original inputs with security against malicious adversaries.

Simply exchanging the hashes and doing local comparison is problematic, because this may reveal

Input to P_1: h_1; decryption key sk_{P_1}.
Input to P_2: h_2.
Public inputs: Public key pk_{P_1}.
Output to P_1: true if $h_1 = h_2$; false otherwise.
Output to P_2: $\perp$.
Execution:

1. P_1 sends to P_2 $\alpha_0 = [\![-h_1]\!]$.
2. P_2 picks random r, s, computes $(e, h) = ([\![r \times (h_2 - h_1) + s]\!], H_2(s, h_2))$, and sends (e, h) to P_1.
3. P_1 computes $\hat{s} = \mathsf{Dec}(e)$. If $H(\hat{s}, h_1) = h$, P_1 outputs true; otherwise, it outputs false.

Figure 5.5: A one-sided equality-testing protocol.

information on the outputs of the circuit evaluation which is supposed to be hidden from the generator unless the validation check passes. For example, already knowing all the output-wire label pairs, the generator who learns the evaluator's hash can test for candidate output values. Therefore, it is of vital importance to keep the hashes secret throughout the comparison protocol if the equality test fails.

Regarding the equality test, the most straightforward realization is to use a generic garbled-circuit protocol (with malicious security). The inputs to the circuit are the hashes $h_1 = H(\mathbb{W}_A^{v_A} \| \boldsymbol{w}_A)$ and $h_2 = H(\boldsymbol{w}_B \| \mathbb{W}_B^{v_B})$, while the circuit is simply a bunch of bitwise XORs (to compute $h_1 \oplus h_2$) followed by a many-to-1 OR circuit that tests if all bits of $h_1 \oplus h_2$ are zero. This still requires running a full garbled-circuit protocol with malicious security, however, which can be expensive.

Hence, we devised an alternative approach, by viewing an equality test as computing the intersection of two singleton sets. Private set intersection has been widely studied in many contexts, including in the presence of malicious adversaries [37, 45, 28, 57, 58, 31]. We derive our secure equality-test protocol (Figure 5.5) by specializing the ideas of Freedman et al. [37] based on additively homomorphic encryption. The basic protocol enables P_2 to prove to P_1 that he holds an h_2 that is equal to h_1 in a privacy-preserving fashion. This basic protocol will be invoked twice with the parties swapping roles. Under the assumption that h_1, h_2 are independent, random values (of sufficient length), it satisfies the following properties even against a malicious adversary: (1) no information about the honest party's input is leaked to the malicious party, and (2) the malicious party can cause the honest party to output 1 with only negligible probability.

First of all, P_1 sends to P_2 $\alpha_0 = [\![-h_1]\!]$. Then, P_2 computes $e = [\![r \times (h_2 - h_1) + s]\!]$, using the homomorphic properties of the encryption scheme, as follows

$$r * \big((h_2 * \alpha_1) \ + \ \alpha_0\big) \ + \ s * \alpha_1$$

where $\alpha_1 = [\![1]\!]$ while $*$ and $+$ denote homomorphic addition and constant multiplication, respectively. In

addition, P_2 sends $h = H(s, h_2)$. P_1 decrypts the result to recover $\hat{s} = r \times (h_2 - h_1) + s$, which is equal to s in case $h_2 = h_1$ but a random value otherwise. Finally, P_1 checks whether $H(\hat{s}, h_1) = h$.

In contrast to the "malicious-client" protocol by Freedman et al. [37], it is unnecessary for P_1 to verify that P_2 followed the protocol (even though P_1 could). The reason is a consequence of several facts:

(a) P_2 doesn't gain anything from seeing $(\alpha_0, \alpha_1, s_1)$;

(b) it is of P_2's own interest to convince P_1 that $h_1 = h_2$;

(c) the test only passes with negligible probability if P_2 cheats.

We remark that the equality test protocol does not appear to achieve the standard notion of (simulation-based) security against malicious adversaries. Nevertheless, under the assumption that h_1, h_2 are independent, random values (of sufficient length), it does, informally, satisfy the following properties even against a malicious adversary: (1) no information about the honest party's input is leaked to the malicious party, and (2) the malicious party can cause the honest party to output 1 with only negligible probability. We conjecture that our proof of security in Section V can be adapted for equality-testing protocols having these properties.

The protocol satisfies the properties claimed above even when both Alice and Bob are malicious. The informal argument is as follows. To see that Alice's privacy is guaranteed, note that $[\![-h_1]\!]$ hides $-h_1$ thanks to the semantic security offered by the homomorphic encryption scheme. Bob's privacy is also guaranteed by the semantic security of both the homomorphic encryption scheme and the cryptographic hash function (in the random-oracle model).

5.3 Proof of Security

We give a rigorous proof of security for the DualEx protocol following the classic paradigm of comparing the real-world execution of the protocol to an ideal-world execution where a trusted third party evaluates the function on behalf of the parties [40]. The key difference is that here we consider a non-standard ideal world where the adversary is allowed to learn an additional bit of information about the honest party's input.

5.3.1 Definitions

Preliminaries. We use n to denote the security parameter. A function $\mu(\cdot)$ is *negligible* if for every positive polynomial $p(\cdot)$ and all sufficiently large n it holds that $\mu(n) < 1/p(n)$.

A *distribution ensemble* $X = \{X(a, n)\}_{a \in \mathcal{D}_n,\, n \in \mathbb{N}}$ is an infinite sequence of random variables indexed by $a \in \mathcal{D}_n$ and $n \in \mathbb{N}$, where $\mathcal{D}_n$ may depend on n.

Distribution ensembles $X = \{X(a,n)\}_{a\in\mathcal{D}_n,\, n\in\mathbb{N}}$ and $Y = \{Y(a,n)\}_{a\in\mathcal{D}_n,\, n\in\mathbb{N}}$ are *computationally indistinguishable*, denoted $X \stackrel{\text{c}}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every n and every $a \in \mathcal{D}_n$

$$\Big|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]\Big| \leq \mu(n).$$

We consider secure computation of *single-output, deterministic* functions where the two parties wish to compute some (deterministic) function f with Alice providing input x, Bob providing input y, and both parties learning the result $f(x,y)$. We assume f maps two n-bit inputs to an ℓ-bit output.

A two-party protocol for computing a function f is a protocol running in polynomial time and satisfying the following correctness requirement: if Alice begins by holding 1^n and input x, Bob holds 1^n and input y, and the parties run the protocol honestly, then with all but negligible probability each party outputs $f(x,y)$.

Security of protocols. We consider static corruptions by *malicious* adversaries, who may deviate from the protocol in an arbitrary manner. We define security via the standard real/ideal paradigm, with the difference being that we use a weaker version of the usual ideal world. Specifically, in the standard formulation of the ideal world there is a trusted entity who receives inputs x and y from the two parties, respectively, and returns $f(x,y)$ to both parties. (We ignore for now the issue of fairness, but discuss a limited form of fairness in Section 5.4.) In contrast, here we consider an ideal world where a malicious party sends its input along with an *arbitrary* boolean function g, and learns $g(x,y)$ in addition to $f(x,y)$. (The honest party still learns only $f(x,y)$.) Note that in this weaker ideal model, *correctness* and *input independence* still hold: that is, the honest party's output still corresponds to $f(x,y)$ for some legitimate inputs x and y, and the adversary's input is independent of the honest party's input. *Privacy* of the honest party's input also holds, modulo a single additional bit that the adversary is allowed to learn.

Execution in the real model. We first consider the real model in which a two-party protocol Π is executed by Alice and Bob (and there is no trusted party). In this case, the adversary $\mathcal{A}$ gets the inputs of the corrupted party and arbitrary auxiliary input aux and then starts running the protocol, sending all messages on behalf of the corrupted party using an arbitrary polynomial-time strategy. The honest party follows the instructions of Π.

Let Π be a two-party protocol computing f. Let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine with auxiliary input aux. We let $\text{VIEW}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n)$ be the random variable denoting the entire view of the adversary following an execution of Π, where Alice holds input x and 1^n, and Bob holds input y and 1^n. Let $\text{OUT}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n)$ be the random variable denoting the output of the honest party after this

execution of the protocol. Set

$$\text{REAL}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n) \stackrel{\text{def}}{=} \left(\text{VIEW}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n),\ \text{OUT}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n)\right).$$

Execution in our ideal model. Here we describe the ideal model where the adversary may obtain one additional bit of information about the honest party's input. The parties are Alice and Bob, and there is an adversary $\mathcal{A}$ who has corrupted one of them. An ideal execution for the computation of f proceeds as follows:

Inputs: Alice and Bob hold 1^n and inputs x and y, respectively; the adversary $\mathcal{A}$ receives an auxiliary input aux.

Send inputs to trusted party: The honest party sends its input to the trusted party. The corrupted party controlled by $\mathcal{A}$ may send any value of its choice. Denote the pair of inputs sent to the trusted party as (x', y'). (We assume that if x' or y' are invalid then the trusted party substitutes some default input.) In addition, the adversary sends an arbitrary boolean function g to the trusted party.

Trusted party sends output: The trusted party computes $f(x', y')$ and $g(x', y')$, and gives both these values to the adversary. The adversary may at this point tell the trusted party to stop, in which case the honest party is given $\perp$. Otherwise, the adversary may tell the trusted party to $\mathsf{continue}$, in which case the honest party is given $f(x', y')$. (As usual for two-party computation with malicious adversaries, it is impossible to guarantee complete fairness and we follow the usual convention of giving up on fairness altogether in the ideal world.)

Outputs: The honest party outputs whatever it was sent by the trusted party; $\mathcal{A}$ outputs an arbitrary function of its view.

We let $\text{OUT}^{\mathcal{A}}_{f,\mathcal{A}(\mathsf{aux})}(x,y,n)$ (resp., $\text{OUT}^{\mathsf{hon}}_{f,\mathcal{A}(\mathsf{aux})}(x,y,n)$) be the random variable denoting the output of $\mathcal{A}$ (resp., the honest party) following an execution in the ideal model as described above. Set

$$\text{IDEAL}_{f,\mathcal{A}(\mathsf{aux})}(x,y,n) \stackrel{\text{def}}{=} \left(\text{OUT}^{\mathcal{A}}_{f,\mathcal{A}(\mathsf{aux})}(x,y,n),\ \text{OUT}^{\mathsf{hon}}_{f,\mathcal{A}(\mathsf{aux})}(x,y,n)\right).$$

Definition 1 Let f, Π be as above. Protocol Π is said to *securely compute f with 1-bit leakage* if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that

$$\left\{\text{IDEAL}_{f,\mathcal{S}(\mathsf{aux})}(x,y,n)\right\}_{x,y,\mathsf{aux}\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n)\right\}_{x,y,\mathsf{aux}\in\{0,1\}^*}$$

Remark. In the proof of security for our protocol, we consider a slight modification of the ideal model described above: namely, the adversary is allowed to *adaptively* choose g after learning $f(x', y')$. Although this may appear to be weaker than the ideal model described above (in that the adversary is stronger), in fact the models are identical. To see this, fix some adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ in the "adaptive" ideal world, where $\mathcal{A}_1$ denotes the initial phase of the adversary (where the adversary decides what input to send to the trusted party) and $\mathcal{A}_2$ denotes the second phase of the adversary (where, after observing $f(x', y')$, the adversary specifies g). We can construct an adversary $\mathcal{A}'$ in the "non-adaptive" ideal world who learns the same information: $\mathcal{A}'$ runs $\mathcal{A}_1$ to determine what input to send, and also submits a boolean function g' defined as follows: $g'(x', y')$ runs $\mathcal{A}_2(f(x', y'))$ to obtain a boolean function g; the output is $g(x', y')$.

5.3.2 Proof of Security

We assume the DualEx protocol runs in a *hybrid world* where the parties are given access to trusted entities computing two functions: oblivious transfer and equality testing. (These trusted entities operate according to the usual ideal-world model where there is no additional one-bit leakage.) We show that the DualEx protocol securely computes f with one-bit leakage in this hybrid model. It follows from standard composition theorems [22] that the DualEx protocol securely computes f with one-bit leakage if the trusted entities are replaced by secure protocols (achieving the standard security definition against malicious adversaries).

Theorem 1. *If the garbled-circuit construction is secure against semi-honest adversaries and H is modeled as a random oracle, then the DualEx protocol securely computes f with one-bit leakage in the hybrid world described above.*

Proof. Let $\mathcal{A}$ denote an adversary attacking the protocol in a hybrid world where the parties have access to trusted entities computing oblivious transfer and equality testing. We assume that $\mathcal{A}$ corrupts Bob, though the proof is symmetric in the other case. We show that we can construct an adversary $\mathcal{S}$, running in our ideal world where the parties have access to a trusted entity computing f, that has the same effect as $\mathcal{A}$ in the hybrid world.

Construct $\mathcal{S}$ as follows:

1. $\mathcal{S}$, given inputs y and aux, runs $\mathcal{A}$ on the same inputs. It then simulates the first-stage oblivious transfers as follows: for the i^{th} oblivious transfer, $\mathcal{S}$ receives $\mathcal{A}$'s input bit y'_i and returns a random "input-wire label" w_i to $\mathcal{A}$.

2. $\mathcal{S}$ sets $y' = y'_1 \cdots y'_n$ and sends y' to the trusted entity computing f. It receives in return an output v_B.

3. $\mathcal{S}$ chooses random output-wire labels

$$\mathbb{W}_A^{v_B} \stackrel{\text{def}}{=} (w_{A1}^{v_{B1}}, \cdots, w_{A\ell}^{v_{B\ell}}).$$

Then, in the usual way (e.g., [69]), $\mathcal{S}$ gives to $\mathcal{A}$ a simulated garbled circuit constructed in such a way that the output-wire labels learned by $\mathcal{A}$ (given the input-wire labels chosen by $\mathcal{S}$ in the first step) will be precisely $\mathbb{W}_A^{v_B}$. Additionally, $\mathcal{S}$ chooses random $w_{A1}^{\bar{v}_{B1}}, \ldots, w_{A\ell}^{\bar{v}_{B\ell}}$, defines

$$\mathbb{W}_A = \begin{pmatrix} w_{A1}^0 & \cdots & w_{A\ell}^0 \\ w_{A1}^1 & \cdots & w_{A\ell}^1 \end{pmatrix},$$

and gives

$$\begin{pmatrix} H(w_{A1}^0) & \cdots & H(w_{A\ell}^0) \\ H(w_{A1}^1) & \cdots & H(w_{A\ell}^1) \end{pmatrix}$$

to $\mathcal{A}$. (The notation $H(\cdot)$ just means that $\mathcal{S}$ simulates a random function on the given inputs.) This completes the simulation of the first stage of the protocol.

4. Next, $\mathcal{S}$ simulates the second-stage oblivious transfers by simply recording, for all i, the "input-wire labels" (w_i^0, w_i^1) used by $\mathcal{A}$ in the i^{th} oblivious transfer. $\mathcal{A}$ then sends its second-phase message (which contains a garbled circuit, input-wire labels corresponding to its own input, and hashes of the output-wire labels).

5. Finally, $\mathcal{A}$ submits some input $\boldsymbol{w}_B \| \boldsymbol{w}'_B$ for the equality test. $\mathcal{S}$ then defines the following boolean function g (that depends on several values defined above):

   (a) On input $x, y \in \{0,1\}^n$, use the bits of x as selector bits to define "input-wire labels" $w_1^{x_1}, \ldots, w_n^{x_n}$. Then, run stage 2 of the protocol exactly as an honest Alice would to obtain $v_A \in \{0,1\}^\ell$ and $\boldsymbol{w}_A$. (In particular, if some error is detected then random values are used for v_A and $\boldsymbol{w}_A$. These random values can be chosen by $\mathcal{S}$ in advance and "hard coded" into g.)

   (b) Return 1 if $\mathbb{W}_A^{v_A} \| \boldsymbol{w}_A$ is equal to $\boldsymbol{w}_B \| \boldsymbol{w}'_B$; otherwise, return 0.

   $\mathcal{S}$ sends g to the trusted party, receives a bit z in return, and gives z to $\mathcal{A}$.

6. If $z = 0$ or $\mathcal{A}$ aborts, then $\mathcal{S}$ sends stop to the trusted entity. Otherwise, $\mathcal{S}$ sends continue. In either case, $\mathcal{S}$ then outputs the entire view of $\mathcal{A}$ and halts.

To complete the proof, we need to show that

$$\left\{\text{IDEAL}_{f,\mathcal{S}(\mathsf{aux})}(x,y,n)\right\}_{x,y,\mathsf{aux}\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\Pi,\mathcal{A}(\mathsf{aux})}(x,y,n)\right\}_{x,y,\mathsf{aux}\in\{0,1\}^*}$$

(where, above, the second distribution refers to the execution of DualEx in the hybrid world where the parties have access to trusted entities computing oblivious transfer and equality). This is fairly straightforward since there are only two differences between the distribution ensembles:

1. In the real world the garbled circuit sent by Alice to $\mathcal{A}$ is constructed correctly based on Alice's input x, while in the ideal world the garbled circuit is simulated based on the input-wire values given to $\mathcal{A}$ and the output v_B obtained from the trusted entity computing f.

2. In the real world the output of the honest Alice when the equality test succeeds is v_A, whereas in the ideal world it is v_B (since v_B is the value sent to Alice by the trusted entity computing f).

Computational indistinguishability of the first change follows from standard security proofs for Yao's garbled-circuit construction [69]. For the second difference, the probability (in the ideal world) that the equality test succeeds and $v_B \neq v_A$ is negligible. The only way this could occur is if $\mathcal{A}$ is able to guess at least one value $w_{Ai}^{\bar{v}_{Bi}}$; but, the only information $\mathcal{A}$ has about any such value is $H(w_{Ai}^{\bar{v}_{Bi}})$. Thus, $\mathcal{A}$ cannot guess any such value except with negligible probability. □

5.4 Enhancements

One problem with the basic DualEx protocol is that it allows the attacker to learn the output of $f(x,y)$ even when cheating since the output is revealed before the equality test. Consequently, this advantage for adversaries could actually encourage participants to cheat and would be unacceptable in many scenarios.

In this section, we present two heuristic enhancements that aim at mitigating the problem. The first enhancement, called *progressive revelation*, is the most straightforward and guarantees that the adversary has can only learn one more bit of the output than the honest party. The second enhancement, we call *DualEx-based equality test*, ensures the outputs are revealed only after the equality check passes. Note that since the two enhancements are orthogonal, they can be combined to construct a improved DualEx protocol that benefits from both.

In both enhancements, to prevent early revelation of outputs we change the output revelation process in the basic DualEx protocol by replacing the final step in the garbled circuit sub-protocol (execution step 3 from Figure 5.3) with a step that just outputs the wire labels without decoding their semantic values:

3. P_1 outputs $\mathbb{W}_1$ that it produced when generating the circuit, while P_2 outputs $\boldsymbol{w}_1^{v_2}$ that it obtains from circuit evaluation.

This changes the output P_2 receives to only include the output-wire labels (and not the underlying bits to which they map).

The delayed revelation modification prevents the semantic values from being learned at the end of each semi-honest protocol execution, and supports the two protocol variations discussed next for revealing the semantic values in a way that ensures a limited notion of fairness.

5.4.1 DualEx-based Equality Test

Our goal is to prevent an adversary from learning the output if it is caught cheating by the equality test. To achieve this, we introduce a pre-emptive secure equality-test protocol that is done before output-wire label interpretation. In addition, compared to the secure equality test used in the basic DualEx protocol (Section 5.2), the test here has to start from output-wire labels (as opposed to be able to use the previously-revealed outputs).

The goal of the pre-emptive equality test is to compute in a privacy-preserving way the predicate $Equal = \mathsf{AND}(Equal_1, Equal_2, \ldots, Equal_\ell)$ in which $Equal_i$ is defined as follows,

$$Equal_i = \begin{cases} 1, & \text{if } \exists \sigma, \text{s.t. } w_{Ai} = w_{Ai}^{\sigma} \textbf{ and } w_{Bi} = w_{Bi}^{\sigma}; \\ 0, & \text{otherwise.} \end{cases}$$

where w_{Ai} (respectively w_{Bi}) is the i^{th} output-wire label Bob (respectively Alice) obtained from circuit evaluation.

The basic idea is to implement *Equal* with a garbled circuit, which ANDs all ℓ wires from ℓ Equal_i circuits. The circuit Equal_i can be implemented as shown in Figure 5.6. The cost of $Equal_i$ is 2σ non-free gates (where σ is the length of a wire label), while the *Equal* circuit requires $2\ell\sigma$ non-free gates. Thus, its cost does not grow with the length of f's inputs nor the f's circuit size (which can be very large).

We could execute the *Equal* circuit with any generic protocol secure against malicious adversaries. Alternatively, the basic DualEx protocol can be employed to keep the overhead low. Note that on average one-bit could be leaked using the DualEx protocol here, but it is a bit about the random nonces used as wire labels, hence does not expose the original private inputs.

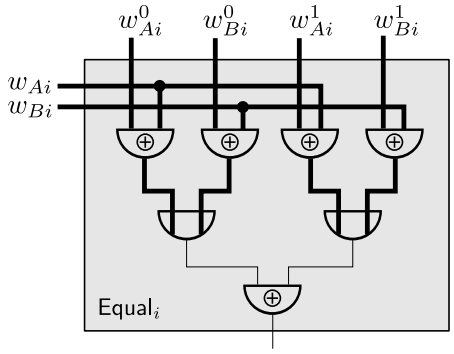


Figure 5.6: Circuit realization of *Equal*$_i$.

5.4.2 Progressive Revelation

The goal of this variation is to reveal the output wires to both parties in a bitwise fashion, until cheating (if there is any) is detected on one output wire. Hence, if the outputs match exactly, both parties will receive the full output at the end of the protocol. If the outputs do not match, both parties will receive the same matching output bits until the first mismatch and the adversary receives at most a single additional mismatched output bit.

The idea resembles that of the gradual release protocols used for exchanging secret keys [18, 30], signing contracts [36], and secure computation [53]. In our scenario, to reveal the i^{th} bit of the output, the parties can securely evaluate a circuit EqualRev$_i$ (Figure 5.7), which tests equality (indicated by v_i) and reveals the output bit (denoted by o_i) at the same time. This circuit looks exactly the same as Equal$_i$ except it has an extra o_i bit which set to 0 if and only if $w_{Ai} = w^0_{Ai}$ and $w_{Bi} = w^0_{Bi}$. The $v_i = 1$ bit implies the o_i bit is indeed valid.

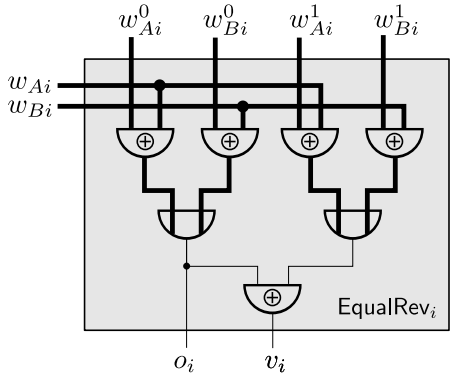


Figure 5.7: Circuit realization of *EqualRev*$_i$.

To further limit an adversary's advantage, we can require the output-wire labels interpretation process to be done in an order that is collectively determined by both parties. For example, let p_a and p_b denote two random permutations solely determined by Alice and Bob, respectively. The two parties will reveal the output bits in the order determined by the permutation $p = p_a \oplus p_b$. Note that to make sure each party

samples its random permutation independent of the other's permutation, p_a and p_b need to be committed before they are revealed.

5.5 Experimental Results

Since there is no need for any party to keep the circuit locally as is required for cut-and-choose, the execution of the garbled circuit sub-protocol (Figure 5.3) can be pipelined as for ordinary semi-honest secure computing protocols. We implement this protocol using our semi-honest model framework (Section 3.4).

In adapting this framework to support dual execution protocols, we observe that Stage 1 and Stage 2 of the dual execution protocol are actually two independent executions of the same semi-honest protocol. Their executions can be overlapped, with both parties simultaneously running the execution where they are the generator and the one where they are the evaluator as two separate threads executing in parallel. Since the workload for the different roles is different, this has additional benefits. Because the generator must perform four encryptions to generate each garbled table, while the evaluator only has to perform a single decryption, the workload for the party acting as the generator is approximately four times that of the evaluator. During normal pipelined execution, this means the circuit evaluator is idle most of the time (Figure 3.1). With simultaneous dual execution, however, both parties have the same total amount of work to do, and nearly all the previously idle time can be fully used.

5.5.1 Experimental Setup

Hardware & Software. The experiments are done on two standard Dell boxes, each equipped with an Intel® Core™ 2 Duo E8400 3GHz processor, 8 GB memory. They are connected with a 100 Mbps LAN. Both boxes are running Linux 3.0.0-12 (64 bit). The JVM version we used is Sun®Java™1.6 SE.

Security Parameters. We use 80-bit nonces to represent wire labels. In our implementation of the Naor-Pinkas OT protocol, we use an order-q cyclic subgroup of $\mathbb{Z}_p^*$ where $|p| = 1024$ and $|q| = 160$. For the implementation of OT extension, we used $k = 80$ and 80-bit symmetric keys. Our security parameters conform to the ultra-short security level recommended by NIST [4].

Applications. We demonstrate the effectiveness of the DualEx protocol with several secure two-party computation applications including private set intersection (Section 4.5), secure edit distance (Section 4.3), and private AES encryption (Section 4.6).

5.5.2 Results

Figure 5.8 summarizes the running time for the three applications running under different settings. The PSI instance is computed over two sets each containing 4096 32-bit numbers using the *Sort-Compare-Shuffle with Waksman Network* (SCS-WN) protocol (Section 4.5.4). The edit distance is calculated from two strings each having 200 8-bit characters. The AES instance is executed in 128-bit key size mode, with 100 iterations. The measurements are the average time over 20 runs of each protocol with randomly generated private inputs (of course, in a secure computation protocol, the running time cannot depend on the actual input values since all operations must be data-independent). We compare our results for DualEx protocols with the results described in Chapter 4.

The measurements include time spent on direct transfer of wire labels, the online phase of oblivious transfer, circuit generation and evaluation, and secure validity test. The time used to initialize the circuit structure and oblivious transfer is not included since these are one-time costs that can be performed off-line.

For symmetric input applications (PSI and ED), we observe the bandwidth cost of dual execution protocols is exactly twice of that for semi-honest protocols. The running time of DualEx protocols running on a dual-core hardware is only slightly higher than that for the corresponding semi-honest protocol. All of the work required for the second execution is essentially done simultaneously with the first execution using the otherwise idle core. The only additional overhead is the very inexpensive equality test at the end of the protocol.

On the other hand, for asymmetric input applications like AES, the dual execution protocol appears to be slower. The reason is that in the semi-honest settings the party holding the message is always designated the circuit generator such that the more expensive oblivious transfers need only to be used for the encryption key (which is shorter than the message). In the DualEx protocol every input bit needs to be obliviously transferred once. Thus, it runs slower than its semi-honest version deployed in favor of using less OTs.

We do not include the time required to compute the "base" OTs (about 12 seconds) in the timing measurements, since this is a one-time, fixed cost that can be pre-computed independent of the actual function to be computed.

Although our implementation is programmed explicitly in two Java threads, we have also run it using a single core for fair comparisons. We used the same software and hardware setup but the processes are confined to be run on a single core using the `taskset` utility command. The corresponding results are shown as the third column in Figure 5.8. Note that the running time is only 42%–47% more than a semi-honest run even if two semi-honest runs are included in the dual execution protocol. Recall that in the semi-honest garbled circuit protocol, the point-and-permute [74] technique sets the workload ratio between the generator

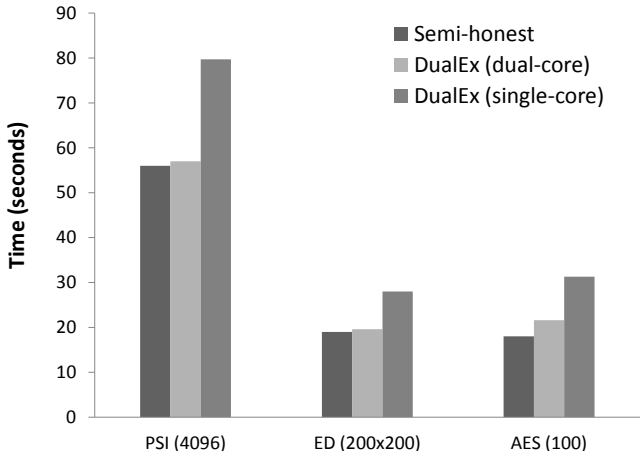


Figure 5.8: Time costs comparing to semi-honest protocols.

and the evaluator to four to one, because the evaluator needs to decrypt only one of the four entries in a garbled truth table. Moreover, garbled-row-reduction [86] optimization brings this ratio down to about 3, since only 3 out of 4 entries in a garbled truth table needs to be transmitted. Therefore, should the overhead of thread-level context switch and interferences are ignored, the slowdown factor of dual execution will be about of 33%. (We assume the network bandwidth is not the bottleneck, which is true on a wired LAN.) Our experimental results actually show that about another 15% of time is lost due to the interference between the two threads.

The scale of the circuits used in our experiments above is already well beyond what has been achieved by state-of-art maliciously-secure secure two-party computation prototypes. However, to fully demonstrate the memory efficiency of the dual execution approach, we also report results from running the PSI and edit distance applications on larger problem sizes. The timing results are shown in Figure 5.9 for performing private set intersection on two sets of one million 32-bit values each, and for an edit-distance computation with input DNA sequences (2-bit character) of 2000 and 10000. The performance of DualEx protocols remains very competitive with semi-honest secure computation protocols even for large inputs.

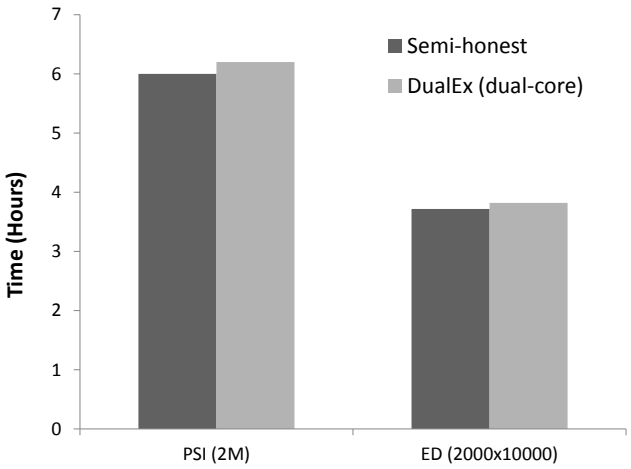


Figure 5.9: Time costs for large scale problems.

Chapter 6

Using Commodity Randomness

The secure computation protocols previously discussed rely heavily on symmetric encryptions to simulate Boolean circuit execution. This leads to running times that are many multiples of what would be required for normal execution. For example, a semi-honest garbled circuit protocol runs approximately a hundred thousand times slower than native executions using a trusted third party, and requires transmitting at least 30 bytes for every non-free binary gate. Costs are orders of magnitude worse under the fully malicious threat model, the additional large constant factor would make the performance figures even more pessimistic.

In this chapter, we present an alternative approach in which we rely on *partially-trusted third parties*. These parties are not trusted with anyone's private input, but only on supplying correlated randomness that satisfies certain properties and not colluding with participants in the computation. In return, we achieve orders-of-magnitude efficiency gains and ensure security in the presence of malicious participants without significant overhead. Our approach follows the commodity-based cryptography paradigm, where the third-party is only trusted for generating and distributing random numbers satisfying certain constraints. The third-party does not receive any sensitive data. Since all randomness is generated independently of the parties' private data, all the random numbers needed can be pre-computed off-line.

6.1 Related Work

Beaver introduced the commodity-based cryptographic paradigm to more efficiently solve secure computation problems and in order to avoid relying on unproven hardness assumptions [9, 10]. Under the commodity-based cryptography model, participating parties retrieve commodity randomness from partially trusted third parties. These third-parties are not trusted with any private data, but only to provide correlated randomness. Building cryptographic primitives around the correlated randomness, such as oblivious transfer [9] allows protocol

designers to build more secure and efficient schemes. However, even though the existence of OT implies arbitrary secure computation, the practical performance benefit remains unclear and we are not aware of any prototype systems built on this theory. Our work is the first prototype system for commodity-based secure two-party computation.

Commodity-based cryptography has been leveraged to build specialized protocols to create more complex secure functions including numerical comparison [21], scalar product [34, 96], k-nearest neighbor classification [109], integer division [95], and oblivious polynomial evaluation [99]. Vaidya and Clifton proposed a general purpose protocol under this paradigm without implementation [101] that is not obviously more efficient than state-of-the-art two-party secure computation protocols (e.g. garbled circuits). In an empirical study, Wang et al. [104] found the commodity-based secure product protocol [34] sufficient to use as a building block in efficient general secure computation protocols. In our work, we provide an implemented general protocol for secure computation using commodity cryptographic primitives that outperforms traditional methods such as garbled circuits.

6.1.1 The NNOB Protocol

The protocol is built around XOR-based secret sharing (i.e., every bit x is divided into x_1 and x_2 such that $x = x_1 \oplus x_2$). It prevents parties from cheating by requiring the secret share holder to always be able to authenticate its share with a share-specific message authentication code (MAC). More specifically, a bit b held by the left-party is represented with a left authenticated bit $\langle b| = (b, K, M)$, where $M = \Delta_L \oplus bK$, $M, K, \in \{0,1\}^n$ are randomly sampled for every bit b while $\Delta_L \in \{0,1\}^n$ is a global nonce that is only known by the right-party). A left authenticated bit $\langle b|$ is always distributed between the two parties so that the left-party has (b, M) whereas the right-party has K. Symmetrically, the right authenticated bit $|b\rangle$ can be defined based on Δ_R, the global secret bit string only known by the left-party. Every binary signal $x = x_1 \oplus x_2$ in the circuit is secret-shared by the two parties, who actually hold $\langle x_1|$ and $|x_2\rangle$ accordingly. Nielsen et al. devised a protocol to compute both AND (i.e., to derive $\langle z_1|$ and $|z_2\rangle$ directly from $\langle x_1|$, $|x_2\rangle$, $\langle y_1|$, $|y_2\rangle$ such that $z = x \wedge y$ where $x = x_1 \oplus x_2, y = y_1 \oplus y_2, z = z_1 \oplus z_2$) and XOR (i.e., to derive $\langle z_1|$ and $|z_2\rangle$ directly from $\langle x_1|$, $|x_2\rangle$, $\langle y_1|$, $|y_2\rangle$ such that $z = x \oplus y$ where $x = x_1 \oplus x_2, y = y_1 \oplus y_2, z = z_1 \oplus z_2$).

Authenticated Bit. An aBit $\langle x\rangle$ is defined as the pair $(\langle x_1|, |x_2\rangle)$ where $x_1 \oplus x_2 = x$. Note $\langle x_1|$ and $|x_2\rangle$ are always joinly held by the two parties.

Randomness Server. The NNOB scheme instantiates this conceptual randomness server with expensive cryptographic protocols based on oblivious transfers. However, in our case, a third party assumes this task of the randomness server. The trusted randomness server S needs to first randomly generate secret bit strings

Δ_L, Δ_R and distributes them to the right-party and left-party, respectively, via a secure channel. Then, S is responsible to supply primitive randomness constrained in one of the following ways (all messages are sent over secure channels):

Left aBit $\langle b|$: S randomly samples $(b, M, K) \in \{0,1\} \times \{0,1\}^n \times \{0,1\}^n$ such that $M = K + b\Delta_L$. It delivers (b, M) to the left party and K to the right party.

Right aBit $|b\rangle$: S randomly samples $(b, M, K) \in \{0,1\} \times \{0,1\}^n \times \{0,1\}^n$ such that $M = K + b\Delta_R$. It delivers (b, M) to the right party and K to the left party.

Left aAND S randomly generates $\langle x|$, $\langle y|$, $\langle z|$ such that $x = yz$. $\langle x|$, $\langle y|$, $\langle z|$ are distributed properly according to the definition of aBit.

Right aAND Symmetrically defined based on **Left aAND**.

Left aOT S randomly generates $\langle b_0|$, $\langle b_1|$, $|c\rangle$, $|z\rangle$ such that $z = b_c$. $\langle b_0|$, $\langle b_1|$, $|c\rangle$, $|z\rangle$ are distributed properly according to the definition of aBit.

Right aOT Symmetrically defined based on **Left aOT**.

Computation over aBits. The following computation involving authenticated bits can be performed by the two computing parties.

XOR aBits By $\langle z| = \langle x| \oplus \langle y|$ we mean, that the left party, who has (x, M_x) and (y, M_y), computes (z, M_z) by $z = x \oplus y$ and $M_z = M_x \oplus M_y$; whereas the right party, who has K_x and K_y, computes K_z by $K_z = K_x \oplus K_y$. XOR-ing two right aBits can be defined symmetrically.

AND public bits By $z = b\langle x|$ we mean, that the left party, who has (x, M_x) and b, computes (z, M_z) by $z = bx$ and $M_z = bM_x$; whereas the right party, who has K_x and b, computes K_z by $K_z = bK_x$. AND-ing a publicly known constant with a right aBits can be defined symmetrically.

Revelation For any authenticated bit (b, M, K), revealing b means the party who holds (b, M) can sends b, M to its peer, who verifies $M = \Delta_* + bK$ where Δ_* is set to either Δ_L or Δ_R, depending on the specific scenario. Note the main cost of the secure computation protocol is due to the revelation operation. For a standalone AND gate, it requires 3 rounds of messages. However, we implemented the protocol so that the third round of AND gates in the current layer is combined with the first round of the AND gates in the next layer. Therefore, only 2 (non-amortizable) rounds are needed for each layer of AND gates.

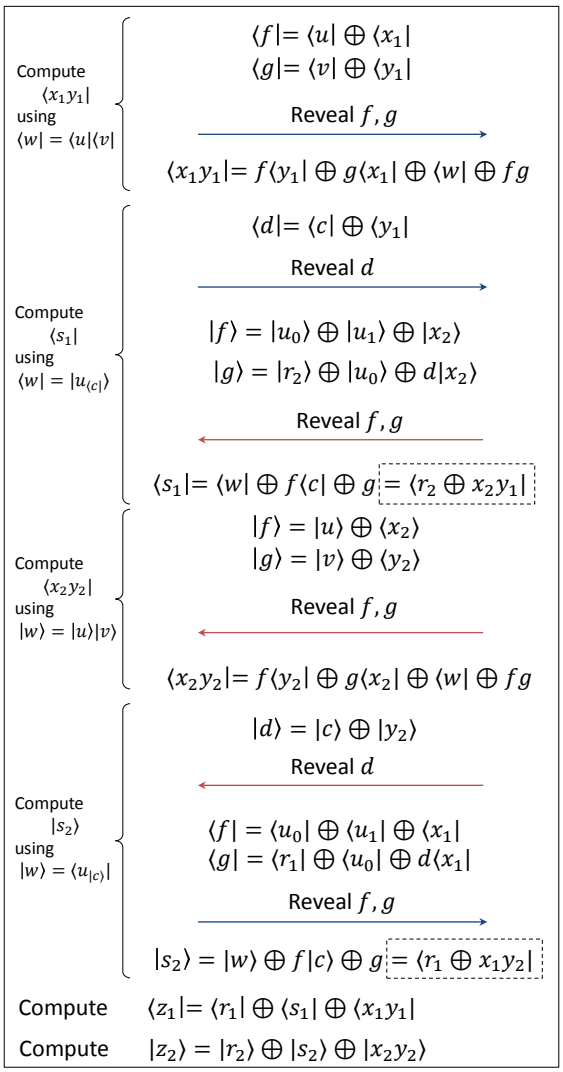


Figure 6.1: The AND protocol

Based on the primitives given so far, it is possible to compute $\langle z\rangle = \langle x\rangle \oplus \langle y\rangle$ and $\langle z\rangle = \langle x\rangle \wedge \langle y\rangle$. Computing the XOR is straightforward — each party only needs to XOR their own entries of the shard aBits because

$$\begin{aligned} \langle x\rangle \oplus \langle y\rangle &= (\langle x_1|, |x_2\rangle) \oplus (\langle y_1|, |y_2\rangle) \\ &= (\langle x_1| \oplus \langle y_1|, |x_2\rangle \oplus \langle y_2|). \end{aligned}$$

However, AND-ing two authenticated bits are rather complicated. The AND protocol is given in Figure 6.1, which is a revised presentation based on the paper by Nielsen et al. [82, Fig. 3].

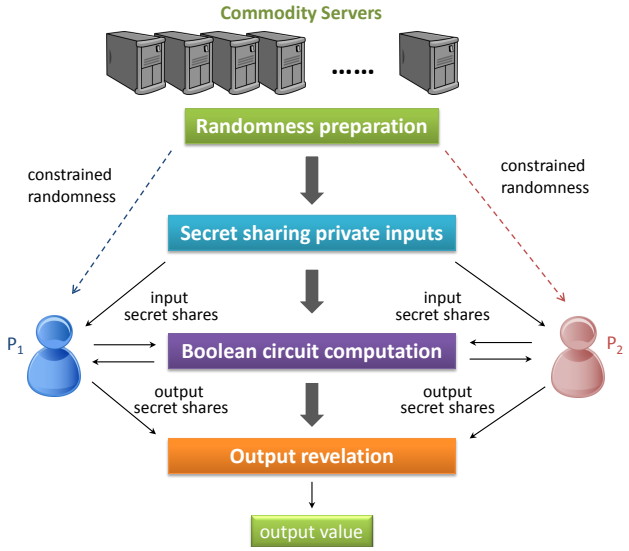


Figure 6.2: Protocol overview

6.2 Overview of Protocols

Conceptually, our protocols can be divided into four phases, as depicted in Figure 6.2. The *randomness preparation* phase can be carried out offline independent of the private inputs. The two parties begin the computation by secret-sharing their private inputs. With the secret shares, they proceed to compute the designated function using a Boolean circuit. Finally, the two parties exchange their shares of the final outputs to accomplish the overall task.

Notably, the collaborative Boolean circuit challenging and time-consuming aspect of the process. The protocol described in Section 6.3 provides a very efficient way to do this in the semi-honest model. This protocol does not provide security against malicious adversaries, however. Section 6.1.1 describes a protocol for the malicious model, built on the scheme by Nielsen et al. [82] but taking advantage of the commodity randomness to provide much better performance.

The two commodity-based protocols differ in the properties the commodity server must provide. For the semi-honest protocol, it is possible to distribute the randomness generation across multiple servers to provide security even if all but one of the servers are colluding with the other participant (Section 6.3.5). However, for the malicious protocol, it is an open problem how randomness provided by multiple servers can be leveraged to thwart collusion.

Input to P_1: private input x', y'.
Input to P_2: private input x'', y''.
Such that $x' + x'' = x,\ y' + y'' = y$.

Commodity random input to P_1: a', b', c'.
Commodity random input to P_2: a'', b'', c''.
Such that $c' + c'' = (a' + a'')(b' + b'')$.
(assuming $a = a' + a'', b = b' + b''$)

Output to P_1: z'.
Output to P_2: z''.
Such that $z' + z'' = x \cdot y$.

Execution:

1. P_1 sends $x' + a'$ and $y' + b'$.
2. P_2 sends $x'' + a''$ and $y'' + b''$.
3. P_1 outputs
$$z' = (x + a)(y + b) - (x + a)b' - (y + b)a' + c'.$$
4. P_2 outputs $z'' = -(x + a)b'' - (y + b)a'' + c''$.

Figure 6.3: The secure product protocol

6.3 Semi-Honest Protocol

We begin the description of the semi-honest protocol with a secure product protocol (Section 6.3.1), in which the randomness preparation step is also given. Next, the subsequent three major phases (shown in Figure 6.2) are discussed sequentially. We conclude this section with techniques for using multiple commodity servers.

6.3.1 The Building Block

The building block of our general secure computation protocol is Beaver's secure product protocol [8] (see Figure 6.3). Given two triples (a', b', c') and (a'', b'', c'') of random elements in $\mathbb{Z}_p$, satisfying the constraint $c' + c'' = (a' + a'')(b' + b'')$, appropriately distributed to P_1 and P_2, the protocol securely computes the shares of $x \cdot y$, where x, y are secretly shared inputs.

This protocol is directly applied to securely compute binary AND gates, assuming the $\mathbb{Z}_2$ context. We describe how to leverage this primitive to achieve arbitrary computation in Section 6.3.3.

Field Choice. This protocol works under any finite field, but we explicitly choose the field $\mathbb{Z}_2$ for simplicity and efficiency. We call it $\mathbb{Z}_2$-based protocol in the rest of the chapter. Its performance is discussed in Section 6.6.

Efficiency. Though each secure dot-product protocol execution requires two communication rounds, only 4 bits of data are transferred in total (assuming $\mathbb{Z}_2$ is used). Additionally, the computational cost of the protocol is extremely low compared to other secure computation approaches (e.g., garbled-circuits) that make heavy use of cryptographic operations.

Randomness Generation. The generation and distribution of the correlated randomness can be completed in a preparation phase independent of the parties' respective private inputs. The amount of required randomness is linear in terms of the number of binary gates needed in the computation (precisely one set of random numbers per binary gate). This preparation phase can be batched offline because the amount of randomness required is both small and pre-computable.

Remove the Commodity Server. The commodity random tuples $(a', a'', b', b'', c', c'')$ can also be obtained without the commodity server. This can be done by invoking a 1-out-of-4 oblivious transfer protocol. In the OT, P_2, who is the receiver, uses 2 randomly generated bits a'', b'' to retrieve the bit c''; while P_1, the sender, prepares four bits of message to send according to the constraint using his own random bits a', b', c' along with all four possible choices. Unfortunately, the performance won't beat that of garbled circuits since it needs 8 encryptions per AND gate even if OT extension is used, whereas an AND in GC uses only 4.

6.3.2 Secret Sharing

The purpose of this stage is to enable the two parties to divide their private inputs into random secret shares, and distribute the secret shares between the two parties. Like the oblivious transfer stage of the garbled circuit protocol, this stage prepares the inputs to the Boolean circuit execution without revealing either participant's private data. However, here it can be achieved much cheaper than running an oblivious transfer protocol.

Let P_1's secret input be $\boldsymbol{x}$ and P_2's be $\boldsymbol{y}$, where $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$. The two participants wish to compute $f(\boldsymbol{x}, \boldsymbol{y})$ for a function f previously agreed upon. To protect its private inputs, party P_1 divides $\boldsymbol{x}$ into secret shares as follows.

1. For every bit x_i in $\boldsymbol{x}$, P_1 samples $x_i' \leftarrow \mathbb{Z}_2$;
2. P_1 sends $x_i'' = x_i - x_i'$ to P_2.

Thus, P_1 knows x_i' and P_2 knows x_i'', where $x_i' + x_i'' = x_i$. Symmetrically, P_2 will go through similar steps with its private input $\boldsymbol{y}$, such that at the end of this stage, P_1 knows y_i' and P_2 knows y_i'' where $y_i' + y_i'' = y_i$. At the next stage, P_1 uses x_i', y_i' (and P_2 uses x_i'', y_i'',) as its inputs to the Boolean circuits.

Essentially, this stage prepares the two parties for the subsequent circuit execution by generating and distributing the secret shares. Note that we do not use any oblivious transfer mechanisms as required in traditional garbled circuit protocols. No encryption operations are needed, only addition in $\mathbb{Z}_2$.

Input to P_1: private input x', y'.
Input to P_2: private input x'', y''.
Such that $x' + x'' = x,\ y' + y'' = y$.

Output to P_1: z'.
Output to P_2: z''.
Such that $z' + z'' = x \oplus y$.

Execution:

1. P_1 and P_2 invoke the secure product protocol to compute $(x' + x'')(y' + y'')$. As a result, P_1 obtains v' and P_2 gets v'' where $v' + v'' = (x' + x'')(y' + y'')$.
2. P_1 outputs $z' = x' + y' + v'$.
3. P_2 outputs $z'' = x'' + y'' + v''$.

Figure 6.4: The secure XOR protocol

6.3.3 Computing Boolean Circuits Securely

The central idea is to compute the function f with a boolean circuit. To this end, we show secure computation protocols for common Boolean gates including AND, OR, XOR, and NOT. We stress that these protocols are designed to enable gate *chaining* to form an arbitrary circuit.

To associate $\mathbb{Z}_2$ numbers to Boolean values, the natural projection is used. Throughout the following discussion, we assume x' and y' are only known to P_1 while x'' and y'' are only known to P_2, such that $x' + x'' = x$ and $y' + y'' = y$, for $x', x'', y', y'' \in \mathbb{Z}_2$. Unless otherwise explicitly specified, we also assume $x, y \in \mathbb{Z}_2$.

AND. We use the secure product scheme (Figure 6.3) to realize secure AND gates, because $x \wedge y = x \cdot y$, where "$\cdot$" denotes $\mathbb{Z}_2$ multiplication.

NOT. We observe that $\overline{x} = 1 - x$, where "$-$" denotes $\mathbb{Z}_2$ subtraction. Let x' (known to P_1) and x'' (known to P_2) be the secret shares of x. To compute $1 - x = 1 - (x' + x'') = (1 - x') - x''$, P_1 outputs $z' = 1 - x'$ while P_2 outputs $z'' = -x''$. Hence $z' + z'' = \overline{x}$ and NOT gate can be done without using any communication or commodity randomness.

OR. OR gates can be constructed using AND and NOT since for all $A, B \in \{\mathsf{false}, \mathsf{true}\}$ $A \vee B = \overline{\overline{A} \wedge \overline{B}}$. Since NOT is free, OR gates have essentially the same cost as AND gates. (Alternatively, it can be realized based on the observation that $x \vee y = x + y - xy$.

XOR. XOR can be realized by requiring each party XOR their own shares, because in $\mathbb{Z}_2$, $+ = \oplus$, thus $x \oplus y = (x' + x'') + (y' + y'') = (x' + y') + (x'' + y'')$. This is essentially free since no communication is needed.

6.3.4 Result Revelation

To reveal the semantic results, for (and only for) every final output-wire, P_1 and P_2 exchange their secret shares to recover the $\mathbb{Z}_2$ value representing the Boolean signal. This simple step works without issue in the semi-honest threat model, since the private bits inside the protocol are perfectly hidden in the output shares due to the random masks. However, in the fully malicious threat model, the attacker could claim an arbitrary value as its secret share, taking advantage of the honest party.

6.3.5 Multiple Commodity Servers

It can be a strong assumption that a single commodity server never colludes with any computation party. Instead, it is more reasonable to assume not all of n servers collude with the same computation party, where n is a configurable parameter. Using multiple commodity servers allows our protocol to withstand attacks where one party colludes with corrupted commodity servers.

To compute xy ($x, y \in \mathbb{Z}_2$) where x, y are secret inputs from P_1, P_2, respectively, each party first randomly divides their secret input into n shares and then collaboratively computes

$$\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right) = \sum_{i=1,j=1}^{i=n,j=n} (x_i y_j).$$

This allows the participants to use up to n^2 different commodity servers. Since the shares are generated uniformly random, unless all of the commodity servers collude with the same computation party, the private inputs x and y remain perfectly hidden. Because the secure computation of $x_i y_j$ ($i, j \in [1, n]$) can be parallelized, using extra commodity servers doesn't increase network latency, increases the bandwidth by k times, where k is the number of servers involved.

6.4 Malicious Model

Originall, the NNOB protocol was designed for the standard two-party computation paradigm, they used a heavy-weight secure two-party protocol to simulate the randomness server that provides the two parties with sufficient amount of constrained randomness-primitives including authenticated bits (aBit), authenticated ANDs (aAND), and authenticated OT (aOT) that are used in the protocol execution. In contrast, we let the parties obtain those primitives from a trusted third party. We stress that all the optimization techniques presented in Section 6.5 are also critical to run the NNOB protocol efficiently.

Although a commodity-based scheme resistant to malicious adversaries is highly desirable, the adapted NNOB scheme using commodity randomness does not provide performance approaching that of the semi-

honest model commodity-based scheme. From a performance perspective, more communication rounds are needed per AND gate, dramatically increasing the overall run-time. The computation integrity in this protocol is *cryptographically* (instead of *unconditionally*) guaranteed, parameterized by the bit length of the MACs and keys. In addition, it is an open question if the protocol can be modified to use multiple randomness servers to thwart collusion attacks.

6.5 Optimizations

To evaluate our protocols, we built a proof-of-concept system based on our secure two-party computation framework (Section 3.4). Instead of using the traditional garbled circuit technique, we replaced the core gate execution with our commodity-based secure computation protocol for evaluating the primitive logic gates including AND, OR, XOR and NOT. In this sense, our scheme is integrated into the framework so that it readily supports existing two-party secure computing applications implemented with circuits.

We started from straightforward implementation of the idea on our framework. Somehow surprisingly, the performance turns out significant worse (thousands times slower) than garbled circuit protocols even though there is not any expensive cryptographic ciphering operations. More careful analysis reveals that it is because of the large number of communication rounds resulted. We note the actual number of rounds is several times the number of binary non-XOR gates contained in the circuit. Taking into account the per round network latencies (e.g., about 10 μs for local processes, about 400 μs on a LAN, and $1 \sim 100$ ms over the Internet), no practical implementation could afford this much penalty for communication.

Next, we present several optimizations that effectively reduce the overhead of communication. They are: (1) layer-by-layer circuit execution (LayerExec), (2) switching roles to reduce the number of rounds (PckRnd), (3) packing and sending bits instead of integers (PckBits), (4) parallelizing the greater-than circuit (ParaGT), and (5) redundant gate elimination, e.g., for greater-than circuits and muxers, (ElmGts).

6.5.1 Layered circuit execution

The original framework executes all the gates sequentially in the depth-first topological order. However, we note that many interesting problems (e.g., those in $\mathcal{NC}$ [85]) can be computed using very wide but shallow circuits. An intuitive way to leverage the problem structure (inspired by the definition of $\mathcal{NC}$-class) is to use sufficiently many threads to execute all readily executable gates in parallel. But this approach can incur prohibitive overhead in thread management and I/O processing. As an alternative, we modified the original execution flow (depth-first circuit traversal implemented as automatic callbacks) of the framework so that it proceeds in breadth-first order. The gate execution are divided into steps based on the direction of

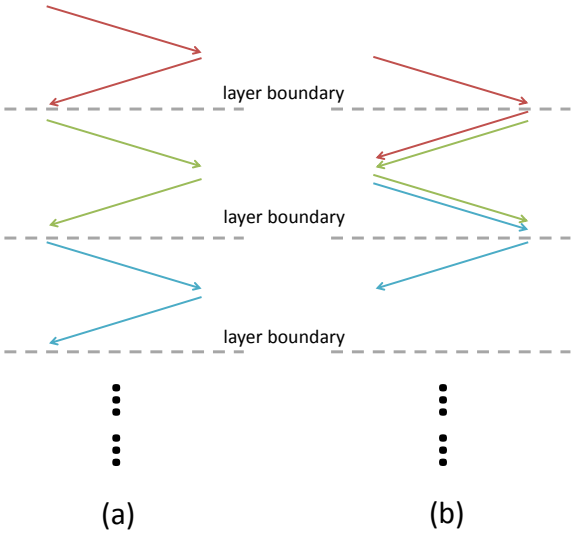


Figure 6.5: Flipping Roles

communications. The finer steps are invoked explicitly in breadth-first order with a queue. This approach only uses a single thread, avoiding all the overhead of extensive multi-threading.

6.5.2 Packing communication rounds

According to the protocol description (Section 6.3), if one party always executes in Alice's role while the other in Bob's role, the protocol requires two communication rounds per dot-product (or per circuit layer when layered execution is employed), as shown in Figure 6.5(a). However, once we alternate the roles of the two parties across layers, n layers can be executed in $n + 2$ rounds. Asymptotically ($n \to \infty$), this reduces the network-latency overhead by half as n approaches ∞ (Figure 6.5(b)).

As another example, a naïve implementation of NNOB protocol according to the steps described in [82, Fig. 3] will result in 6 communication rounds (each reveal incurs 1 round) per AND gate. It can be redued to 3 (as is shown in Figure 6.1) by carefully rearranging the order of some independent messages, such that 2 reveals are done in the first round and another three accomplished in the second.

Packing the bits. To reduce bandwidth, we wrote our own I/O module that automatically packs every eight $\mathbb{Z}_2$ elements into a single byte. This reduces the bandwidth to 1/8 of what it would be using Java's `writeByte` to transfer $\mathbb{Z}_2$ numbers.

6.5.3 Circuit Re-design

The circuit designs described in Chapter 4 are optimized solely for exploiting the free-XOR technique. However, using commodity-based schemes, the depth of the circuit is also very important for producing good

performance. Hence, we want to make the circuit as shallow as possible to reduce network latency. Next, we use the greater-than circuit as an example to study the design space of more efficient circuits along this direction.

Parallelized greater-than. A frequently used basic circuit in secure computing applications is GT_2L_1, which compares 2 L-bit numbers. The state-of-art GT_2L_1 circuit uses the design by Kolesnikov et al. [63, Fig. 6], which incurs L rounds of communication (since it executes L GT_3_2 circuits serially).

To reduce the depth, the greater-than circuit needs a completely different design. Let C_i be the carry-in of the i-th 1-bit comparator, and X_i, Y_i be the two input bits to be compared. Starting from Kolesnikov et al.'s greater-than circuit design [63], we have $C_{i+1} = X_i \oplus \big[(X_i \oplus C_i) \wedge (Y_i \oplus C_i)\big]$, so

$$\begin{aligned} C_{i+1} &= X_i \oplus \big[(X_i \oplus C_i) \wedge (Y_i \oplus C_i)\big] \\ &= X_i \oplus X_iY_i \oplus X_iC_i \oplus Y_iC_i \oplus C_i \\ &= (X_i \oplus X_iY_i) \oplus (X_i \oplus Y_i \oplus 1)C_i \\ &= (X_i\overline{Y_i}) \oplus (X_i \oplus \overline{Y_i})C_i. \end{aligned}$$

Let $G_i = X_i\overline{Y_i}$ and $P_i = X_i \oplus \overline{Y_i}$. The output of GT_2L_1 is

$$\begin{aligned} C_L &= G_{L-1} \oplus P_{L-1}C_{L-1} \\ &= G_{L-1} \oplus P_{L-1}(G_{L-2} \oplus P_{L-2}C_{L-2}) \\ &= G_{L-1} \oplus P_{L-1}G_{L-2} \oplus P_{L-1}P_{L-2}C_{L-2} \\ &= \ldots \text{(keep substituting and expanding the last } C) \ldots \\ &= G_{L-1} \oplus P_{L-1}G_{L-2} \oplus \cdots \oplus P_{L-1}\cdots P_1G_0 \\ &\quad \oplus P_{L-1}\cdots P_0C_0 \\ &= G_{L-1} \oplus P_{L-1}G_{L-2} \oplus \cdots \oplus P_{L-1}\cdots P_1G_0 \end{aligned}$$

because $C_0 = 0$. Note that each G_i and P_i can be computed in one step solely based on X_i and Y_i. In addition, the longest XOR term (computed by AND-ing L signals) can be finished within $\lceil \log L \rceil$ steps using $L-1$ binary ANDs structured as a tree. Therefore, the latency cost of GT_2L_1 is reduced from L to $\log L + 1$ rounds.

Eliminating redundant gates. Note that a naïve translation of the previous C_L equation into a greater-than circuit can be much more costly than necessary, due to repeated computation. For example, the AND

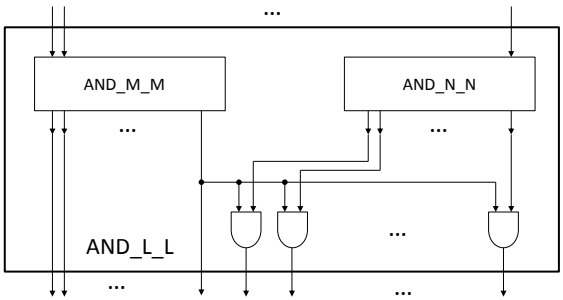


Figure 6.6: The AND_L_L circuit ($M, N < L$)

of P_{L-1} and P_{L-2} is computed $L-2$ times. Asymptotically, such a circuit uses $O(L^2)$ binary AND gates. Our solution is to use an AND_L_L gate instead of L AND_L_1 gates. The AND_L_L gate has L-bit inputs and L-bit outputs, where the i-th output is the AND of all first i input bits. We constructed this circuit in a recursive manner as shown in Figure 6.6, where M is the largest perfect power of 2 smaller than L while $N = L - M$. With the master theorem on recurrence, it is easy to see that the simplification requires only $O(L \log L)$ binary AND gates. Thus, this optimization reduces the number of AND gates from quadratic to linear in L.

6.6 Evaluation

To evaluate the performance of the schemes and understand the impact of different optimizations, we implemented them and measured the timings for several secure computing applications. The experiments are run in two different settings:

LAN Two Dell desktop computers (Intel® Core™ 2 Duo E8400 3GHz, Ubuntu Linux (kernel version 3.0.0), Java 1.6), placed in the same building, are connected via 100 Mbps LAN. The `ping` latency is about 0.5 ms.[1] All the experiments in previous chapters are done in a LAN environment.

Internet Two desktops, sitting 2 miles away, are connected through Internet. One machine is configured as described above, while the other is has Intel® Core™ i7-2600S 2.80GHz, running Windows 7, Java 1.7). In particular, one endpoint (the Windows desktop) uses a wireless connection of 802.11g with WPA2 encryption. The `ping` latency is around 4 ms, 5∼15 times of the LAN setting.

Performance. We compared the performances of commodity-based protocols to traditional semi-honest model garbled circuit based protocol (GC-based) in the *ultra-short* security setting [49] (80-bit wire-labels, $(80, 80)$ as the OT security parameter, SHA-1), using PSI as a case study. Figures 6.7 and 6.8 summarize our

[1] Note the network latency seen in our protocol (running over TCP) is usually much larger than that for the `ping` protocol because `ping` is a much simpler application layer protocol using ICMP.

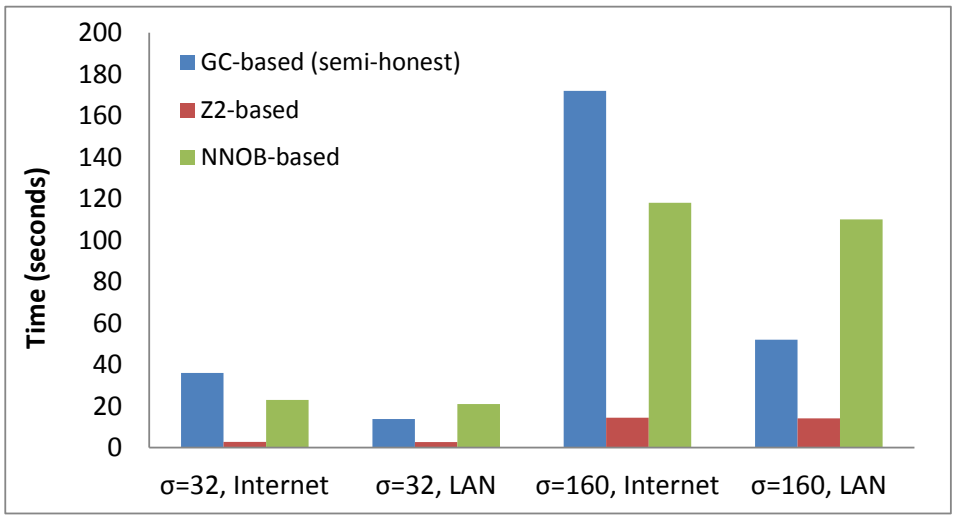


Figure 6.7: Performance Comparison (PSI, $n = 1024$)

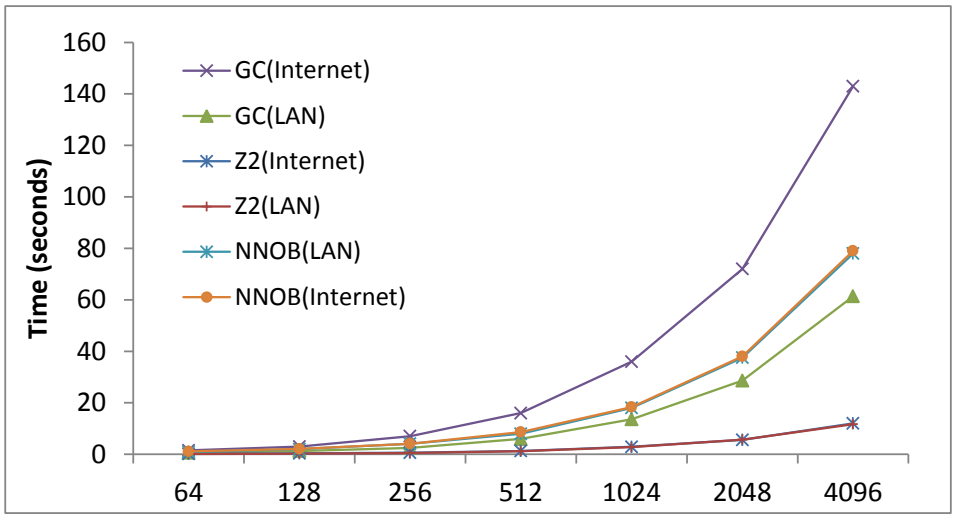


Figure 6.8: Performance Comparison (PSI, $\sigma = 32$)

results. The $\mathbb{Z}_2$-based protocols are about 12 times faster in the Internet setting and about 5 times faster over the LAN. Generally, assuming the circuit from the $\mathcal{NC}$ classes is sufficiently wide, the advantage of $\mathbb{Z}_2$-based protocols becomes more significant as it makes better use of the long network latency by packing more gates into every round-trip.

Regarding the bandwidth, $\mathbb{Z}_2$-based protocols are about 30 times more efficient than the GC-based ones (Figure 6.9). The NNOB-based protocol generates more than 3.5 times of network traffic (Figure 6.9) needed by the $\mathbb{Z}_2$-based scheme, when the amortized MAC checking optimization [82, Section 3] is implemented. However, it is about 8 times slower than the $\mathbb{Z}_2$-based scheme in computing the PSI (Table 6.1). This is due to the computational sophistication of the NNOB-based scheme. For example, the NNOB-based scheme

requires more communication rounds (2 rounds per layer as opposed to 1 for the $\mathbb{Z}_2$-based scheme).

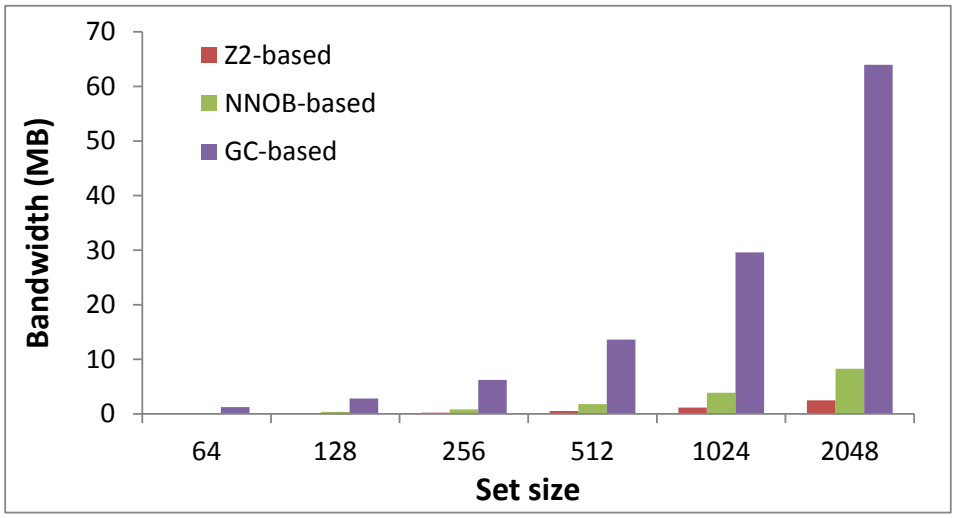


Figure 6.9: Bandwidth Comparison (PSI, $\sigma = 32$)

In higher security settings, the performance of GC-based protocols reduces as the security parameters increase whereas the performance of $\mathbb{Z}_2$-based protocols remains the same since they are already perfectly secure. Hence, the performance gap becomes even bigger.

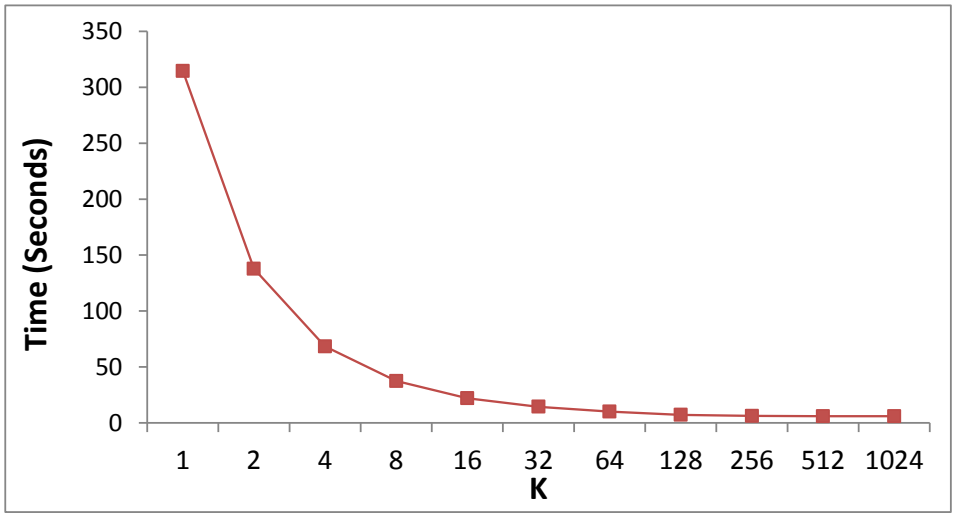


Figure 6.10: Performance impact of circuit width

The performance of commodity-based protocols can be very sensitive to the width of the circuit, especially when network latency is significant. Take $\mathbb{Z}_2$-based protocol as an example, Figure 6.10 shows the impact of circuit width on the bitonic sorting stage ($\sim 85\%$ of the cost of the PSI protocol) in the Internet experiment setting, where K is the number of 2-SORTERs juxtaposed in the circuit execution. Hence, the bigger K is, the more gates can be processed in each round of communication. From the graph, we can deduce that the

		$\mathbb{Z}_2$-based			NNOB-based			GC-based (semi-honest)	
		Time	B/w	Depth	Time	B/w	Depth	Time	B/w
PSI σ=32 n=1024	LAN	2.6 s	1.1 MB	103	17.0 s	3.9 MB	103	13.6 s	29.6 MB
	Internet	2.9 s			18.3 s			36.3 s	
AES	LAN	0.19 s	6.5 KB	640	0.38 s	17.9 KB	640	0.24 s	315.4 KB
	Internet	1.19 s			2.38 s			0.52 s	

Table 6.1: NNOB-based versus secure product-based protocols

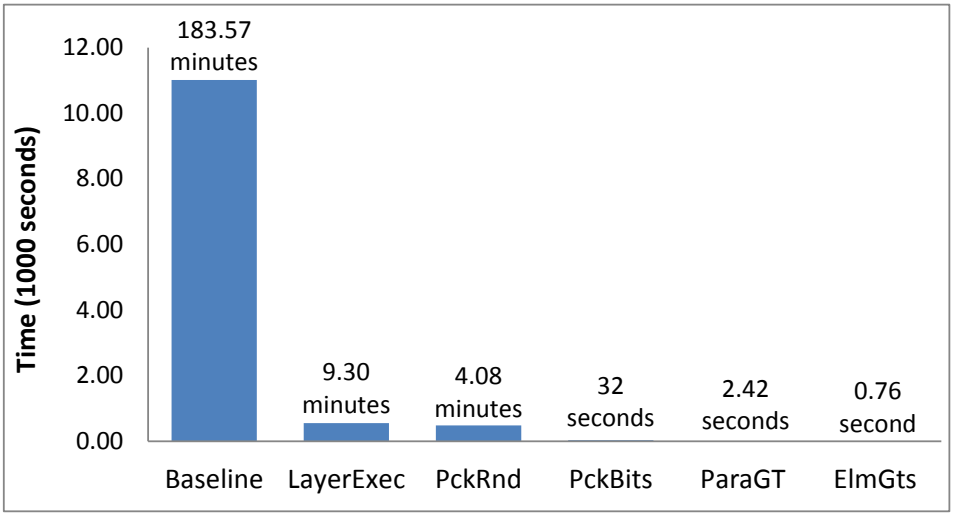
network latency in the Internet setting is filled as K becomes 128 or larger, since it does not become more time-efficient after $K = 128$.

Effectiveness of optimization. Figure 6.11 shows the effectiveness of our optimization techniques, assuming each optimization is applied sequentially over the previous ones in the order given above. We report the optimization impact both in timings (Figure 6.11(a)) and speedups (Figure 6.11(b)). The experiments were done in the Internet setting while computing the private set intersection between two 128 elements (32-bit numbers) sets. Notably, the benefit of reducing communication rounds (PckRnd) is not around 2 as we anticipated, since the network latency is already filled up to a large extent. (PckRnd will bring a speedup of 2 only when latency is the main cost, while at the other extreme when the latency is completely filled up, it could no longer bring any benefits.)

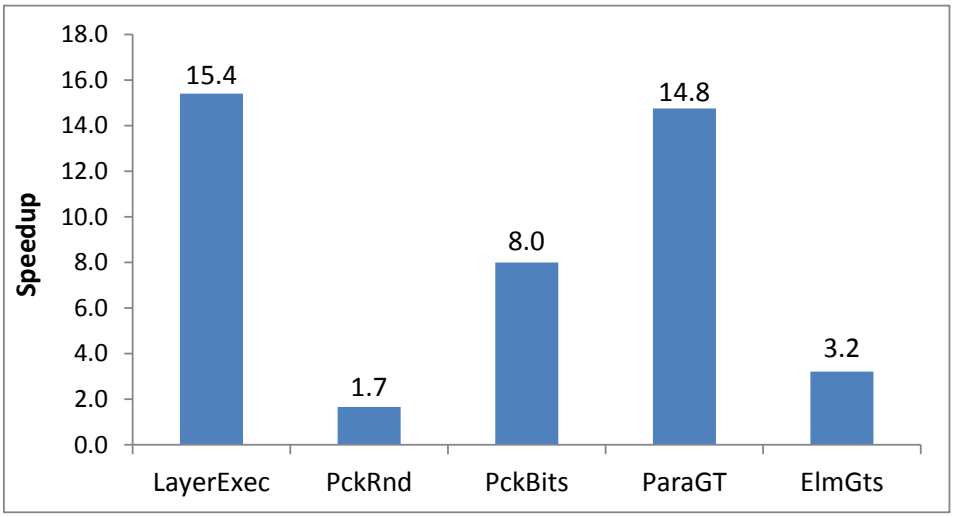
To demonstrate that the circuit depth is not an issue after the optimizations, we show the number of layers in terms of the set size in Figure 6.12. The numbers are taken after all the optimizations, for three different σ (the number of bits to represent a set element) values. We observe that the number of circuit layers grows very slowly with the PSI problem scale in terms of both set size and σ.

AES vs. PSI. Regarding AES, we used the exact same circuit structure (which was optimized for minimizing the number of non-XOR gates) across all three protocols. The results show an example where our approach does not produce good results, due to the naïve reuse of the circuit structure. The circuit is based on a narrow (128-bit width) and deep AES circuit implementation, which greatly limits the advantages of our approach (since they easily become network-latency-bound) but does not affect the semi-honest GC-based scheme (because it is already CPU-bound). Employing an AES design with wide but shallow circuit (e.g., the one proposed by Boyar and Peralta [20]) would improve the performance significantly for $\mathbb{Z}_2$-, and NNOB-based protocols. Thus, the $\mathbb{Z}_2$-based AES protocol is severely disadvantaged due to the large portion of XORs in our AES circuit.

Through this naïve implementation of private AES encryption, we learned that careful application-specific circuit design plays an important role in achieving adequate performance. Compared to the implementation of



(a) time improvements



(b) speedup

Figure 6.11: Effects of optimization (PSI, $n = 128, \sigma = 32$)

AES by Nielsen et al. [82], our layered execution and packing techniques improved the efficiency of executing a single AES circuit without needing to pack hundreds of AES executions in parallel to achieve the amortized low execution time.[2] Comparing our optimized NNOB-based implementation against the recent garbled circuits resistant against malicious adversaries, we see a performance improvement in AES execution of about 500 times [94].

[2]For a single AES execution, the 4 seconds reported in their paper [82] does not include any network latency, as verified via email correspondence with the authors. Notably, the cost from network latency in one AES execution is about 10^{-3} (seconds) $\times$ 2 (rounds) $\times 10^4$ (gates) $= 20$ seconds, so the compute cost measures only a small fraction of the total cost.

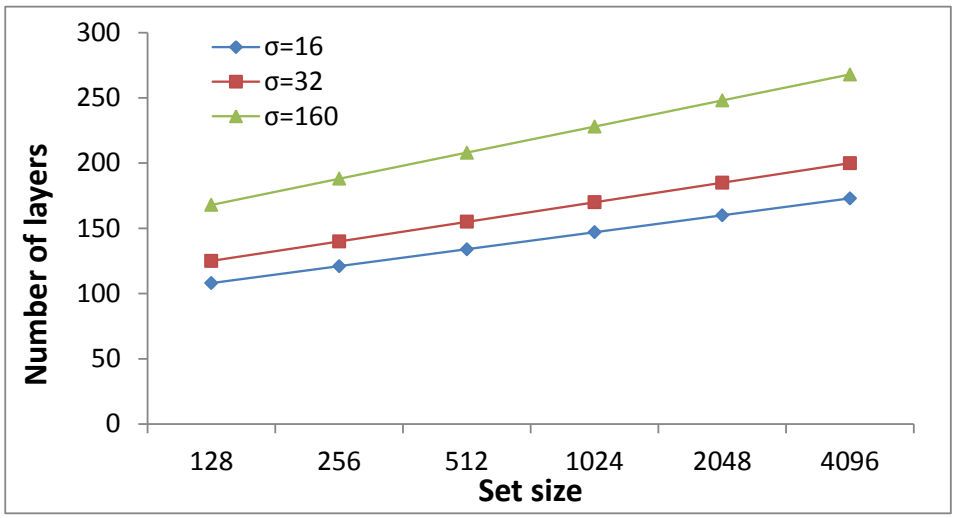


Figure 6.12: Growth rate of circuit layers

6.7 Summary

Commodity-based secure computation protocols rely on third parties to supply correlated random numbers. It serves as a good example of modifying the trust model for useful trade-off between security and efficiency (comparing to the 1-bit leak model that relaxes the threat model to trade security for efficiency). Protocols in this model exhibit very attractive performance figures, especially when active attackers are considered. This trust model will be useful in many scenarios, where computational resource is very limited but weak trust over some third parties can be easily established (e.g., secure computation on mobile devices with correlated randomness from the mobile service providers).

Chapter 7

Conclusion

This chapter begins with a summary of the thesis work (Section 7.1), followed by a recap of the contributions (Section 7.2), and closes with final remarks in Section 7.3.

7.1 Summary

Our work helps to dissipate the misconceptions about the performance and scalability of generic secure two-party computation protocols (which once led to the development of several complex, special-purpose protocols for problems that are better addressed with generic approaches). We demonstrate design optimizations and implementation techniques that enable garbled circuits scale to many large problems, and are practical enough to be competitive with special-purpose protocols.

Our framework enables users to take advantage of low-level circuit design to produce efficient and scalable privacy-preserving protocols. We identified several techniques for producing efficient circuits, and developed a framework for evaluating them efficiently, and demonstrated its impact by implementing several privacy-preserving applications and executing secure computations orders of magnitude larger than has been done before.

We also demonstrate the potential of an alternate approach for security against a malicious adversary, which relaxes the security properties by allowing a single bit of extra information to leak. This relaxation allows us to implement privacy-preserving applications with much stronger security guarantees than semi-honest protocols, but with minimal extra cost. The applications scale to large inputs on commodity machines, including million-input private set intersection.

We explore the commodity-based cryptography approach of secure two-party computation. Our experiments show its potential in making secure computation practical, even in the malicious setting. By

placing fairly limited trust (that the private-input independent constrained-randomness is correctly generated and the commodity servers do not collude) in commodity servers, we obtain substantial gains in efficiency. Commodity-based cryptography could serve as a promising direction in the goal towards widely deployed, privacy-preserving applications.

7.2 Contributions

We developed techniques that enable garbled circuit execution to scale to large applications. These techniques include pipelined execution, reducing circuit width, library-based circuit construction, and garbled/plain hybrid execution. They are built into an integrated software framework that facilitates creating secure two-party computing applications. In the semi-honest threat model, our framework produces protocols that are orders of magnitude faster than has been achieved in previous works.

We presented a concrete design and implementation of protocols in the 1-bit leakage threat model. This model offers much stronger security guarantees than are possible with semi-honest protocols, at minimal extra cost. Specifically, a malicious adversary may learn only a single bit of additional information about the honest party's input. The implementation features a highly efficient mechanism for carrying out the equality test in the presence of malicious adversaries, and incorporates pipelined execution and other efficiency optimizations.

We developed a method for building efficient Boolean-circuit secure two-party computation protocols using commodity randomness. Protocols in this model can run one to many orders of magnitude faster than garbled circuit based schemes, but without overly trusting a third party. Our main contributions here include efficient realization of layered circuit executions and depth-oriented circuit optimizations, which mitigate the performance bottleneck caused by network latencies.

The effectiveness of our framework is evaluated by building several privacy-preserving applications, including genomic analysis, Hamming distance, Euclidean distance, AES, and set intersection. In the semi-honest threat model, we demonstrate secure computation of circuits with over 10^9 gates at a rate of roughly 10μs per garbled gate, which is order-of-magnitude improvements over the best previous implementations. In the 1-bit leak model, we show protocols with performance close to semi-honest settings but providing stronger security guarantees against malicious adversaries. Last, with server assistance for distributing correlated randomness, we show that private set intersection and AES can be computed orders of magnitude faster than the state-of-art maliciously secure protocol implementations.

7.3 Conclusion

We present a step towards the goal of using private data for productive collaboration, in particular, showing that efficient, large-scale secure computations can be built with a reasonable amount of effort for interesting applications. Additional challenges remain before secure computation can be used routinely in practice. We anticipate that wide adoption of secure computation techniques can bring many revolutionary applications that makes (private) information work for the society.

Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computation (STOC)*, 1983.

[2] T. Atkinson, R. Bartak, M. Silaghi, E. Tuleu, and M. Zanker. Private and Efficient Stable Marriages (Matching). *eprint.iacr.org*, 2006.

[3] Y. Aumann and Y. Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *Journal of Cryptology*, 23:281–343, April 2010.

[4] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST special publication 800-57: Recommendation for key management — part 1, March 2007.

[5] M. Barni, T. Bianchi, D. Catalano, M. D. Raimondo, R. D. Labati, P. Faillia, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, and A. Piva. Privacy-Preserving Fingercode Authentication. In *12th ACM Multimedia and Security Workshop*, 2010.

[6] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*. ACM, 1968.

[7] A. Bazen and S. Gerez. Systematic Methods for the Computation of the Directional Fields and Singular Points of Fingerprints. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2002.

[8] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, 1991.

[9] D. Beaver. Commodity-based Cryptography. In *ACM Symposium on Theory Of Computing*, 1997.

[10] D. Beaver. Server-assisted Cryptography. In *New Security Paradigms Workshop*, 1998.

[11] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. In *ACM Symposium on Theory of Computing*, 1990.

[12] M. Bellare and O. Goldreich. On Defining Proofs of Knowledge. In *Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 390–420, London, UK, UK, 1992. Springer-Verlag.

[13] M. Bellare and P. Rogaway. Random Oracles are Practical: a Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, 1993.

[14] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[15] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *ACM Conference on Computer and Communications Security*, 2008.

[16] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *ACM Symposium on Theory of Computing*, 1988.

[17] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[18] M. Blum. How to Exchange (Secret) Keys. *ACM Transactions on Computer Systems*, 1(2):175–193, 1983.

[19] J. Boyar, P. Matthews, and R. Peralta. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology*, 2012.

[20] J. Boyar and R. Peralta. A Depth-16 Circuit for the AES S-box. Cryptology ePrint Archive, 2011. `http://eprint.iacr.org/2011/332`.

[21] C. Cachin. Efficient Private Bidding and Auctions with an Oblivious Third Party. In *ACM conference on Computer and Communications Security*, 1999.

[22] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[23] D. Canright. A Very Compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2005.

[24] C. Chen, R. Veldhuis, T. Kevenaar, and A. Akkermans. Biometric Binary String Generation with Detection Rate Optimized Bit Allocation. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Workshop on Biometrics*, pages 1–7, Los Alamitos, June 2008. IEEE Computer Society Press.

[25] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web Applications via Automatic Partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[26] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX Security Symposium*, 2007.

[27] E. D. Cristofaro and G. Tsudik. Experimenting with Fast Private Set Intersection. In *TRUST*, volume 7344 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2012.

[28] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient Robust Private Set Intersection. In *International Conference on Applied Cryptography and Network Security*, pages 125–142, 2009.

[29] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[30] I. Damgård. Practical and Provably Secure Release of a Secret and Exchange of Signatures. *Journal of Cryptology*, 8(4):201–222, 1995.

[31] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, 2010.

[32] E. De Cristofaro and G. Tsudik. Practical Private Set Intersection Protocols with Linear Complexity. In *Financial Cryptography*, 2010.

[33] Y. G. Desmedt and Y. Frankel. Threshold Cryptosystems. In *CRYPTO*, 1989.

[34] W. Du and Z. Zhan. A Practical Approach to Solve Secure Multi-party Computation Problems. In *Workshop on New Security Paradigms*, 2002.

[35] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-Preserving Face Recognition. In *Privacy Enhancing Technologies*, volume 5672 of *Lecture Notes in Computer Science*, pages 235–253. Springer, 2009.

[36] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[37] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient Private Matching and Set Intersection. In *Eurocrypt*, 2004.

[38] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995.

[39] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust Threshold DSS Signatures. In *EUROCrypt*, 1996.

[40] O. Goldreich. *Foundations of Cryptography, vol. 2: Basic Applications*. Cambridge University Press, 2004.

[41] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. In *ACM Symposium on Theory of Computing*, 1987.

[42] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof-Systems. In *Annual ACM Symposium on Theory of Computing*, pages 291–304, New York, NY, USA, 1985. ACM.

[43] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[44] D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-Combiners via Secure Computation. In *Proceedings of the 5th Conference on Theory of Cryptography*, pages 393–411, Berlin, Heidelberg, 2008. Springer-Verlag.

[45] C. Hazay and Y. Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In *TCC*. Springer-Verlag, 2008.

[46] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[47] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security*, 2010.

[48] S. Henikoff and J. G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. In *Proceedings of the National Academy of Sciences of the United States of America*, 1992.

[49] Y. Huang, D. Evans, and J. Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[50] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.

[51] Y. Huang, J. Katz, and D. Evans. Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution. In *IEEE Symposium on Security and Privacy*, pages 272–284, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[52] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient Privacy-Preserving Biometric Identification. In *NDSS*, 2011.

[53] R. Impagliazzo and M. Yung. Direct Minimum-Knowledge Computations. In *Advances in Cryptology — Crypto*, pages 40–51, 1988.

[54] U. Inria, S. Antipolis, B. Beauquier, B. Beauquier, E. Darrot, E. Darrot, and P. Sloop. On arbitrary Waksman networks and their vulnerability. Research Report 3788, INRIA, 1999.

[55] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, 2003.

[56] A. K. Jain, S. Prabhakar, L. Hong, and S. Pankanti. Filterbank-Based Fingerprint Matching. *IEEE Transactions on Image Processing*, 9:846–859, 2000.

[57] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *Theory of Cryptography Conference*, pages 577–594, 2009.

[58] S. Jarecki and X. Liu. Fast Secure Computation of Set Intersection. In *International Conference on Security and Cryptography for Networks*, pages 418–435, 2010.

[59] S. Jarecki and V. Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In *Eurocrypt*, 2007.

[60] A. Jarrous and B. Pinkas. Secure Hamming Distance Based Computation and Its Applications. In *International Conference on Applied Cryptography and Network Security*, 2009.

[61] S. Jha, L. Kruger, and V. Shmatikov. Towards Practical Privacy for Genomic Computation. In *IEEE Symposium on Security and Privacy*, 2008.

[62] K. V. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. In *Applied Cryptography and Network Security (ACNS)*, 2011.

[63] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *International Conference on Cryptology and Network Security*, 2009.

[64] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages and Programming*, 2008.

[65] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography*, 2008.

[66] P. Li and S. Zdancewic. Downgrading Policies and Relaxed Noninterference. In *ACM Symposium on Principles of Programming Languages*, 2005.

[67] Y. Lindell, E. Oxman, and B. Pinkas. The IPS Compiler: Optimizations, Variants and Concrete Efficiency. In *Advances in Cryptology — Crypto*, pages 259–276, 2011.

[68] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *EUROCRYPT*, 2007.

[69] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[70] Y. Lindell and B. Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In *Theory of Cryptography Conference*, 2011.

[71] Y. Lindell, B. Pinkas, and N. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In *International Conference on Security and Cryptography for Networks*, pages 2–20, 2008.

[72] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[73] P. MacKenzie, A. Oprea, and M. Reiter. Automatic Generation of Two-party Computations. In *ACM Conference on Computer and Communications Security*, 2003.

[74] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — A Secure Two-Party Computation System. In *USENIX Security Symposium*, 2004.

[75] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar. *Handbook of Fingerprint Recognition*. Springer, 2009.

[76] P. Mohassel and M. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *International Conference on Theory and Practice of Public Key Cryptography*, pages 458–473, 2006.

[77] R. Mott. Smith-Waterman Algorithm. In *Encyclopedia of Life Sciences*. John Wiley & Sons, 2005.

[78] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symposium on Principles of Programming Languages*, 1999.

[79] A. C. Myers and B. Liskov. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, October 2000.

[80] M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[81] J. Nielsen and C. Orlandi. LEGO for Two-Party Secure Computation. In *Theory of Cryptography Conference*, 2009.

[82] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. Crypto ePrint Archive, 2011. `http://eprint.iacr.org/2011/091`.

[83] J. D. Nielsen and M. I. Schwartzbach. A Domain-specific Programming Language for Secure Multiparty Computation. In *Workshop on Programming Languages and Analysis for Security*, 2007.

[84] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI: A System for Secure Face Identification. In *IEEE Symposium on Security and Privacy*, 2010.

[85] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[86] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, 2009.

[87] S. Prabhakar and A. Jain. Decision-Level Fusion in Fingerprint Verification. *Pattern Recognition*, 2002.

[88] M. O. Rabin. How to exchange secrets with oblivious transfer. Technical Report 81, Harvard University, 1981.

[89] V. Rajeshirke, J. Nzouonta, and M. Silaghi. Meeting Scheduling with Privacy. `http://www.cs.fit.edu/~msilaghi/pages/secure/meeting-scheduling/`, 2004. (accessed on January 11, 2011).

[90] A. Ross, A. Jain, and J. Reisman. A Hybrid Fingerprint Matcher. *Pattern Recognition*, 2003.

[91] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[92] A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In *IEEE Computer Security Foundations Workshop*, 2005.

[93] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient Privacy-Preserving Face Recognition. In *International Conference on Information Security and Cryptology*, 2009.

[94] C.-H. Shen and A. Shelat. Two-output Secure Computation With Malicious Adversaries. In *EUROCRYPT*, 2011.

[95] C.-H. Shen, J. Zhan, T.-S. Hsu, C.-J. Liau, and D.-W. Wang. Scalar-product based Secure Two-Party Computation. In *International Conference onGranular Computing*, 2008.

[96] C.-H. Shen, J. Zhan, D.-W. Wang, T.-S. Hsu, and C.-J. Liau. Information-Theoretically Secure Number-Product Protocol. In *International Conference on Machine Learning and Cybernetics*, 2007.

[97] M. Silaghi. SMC: Secure Multiparty Computation Language. `http://www.cs.fit.edu/~msilaghi/pages/SMC/tutorial.html`, Nov. 2004. (accessed on January 11, 2011).

[98] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147, 1981.

[99] R. Tonicelli, R. Dowsley, G. Hanaoka, H. Imai, J. Müller-Quade, A. Otsuka, and A. C. A. Nascimento. Information-Theoretically Secure Oblivious Polynomial Evaluation in the Commodity-Based Model. Cryptology ePrint Archive, 2009. `http://eprint.iacr.org/2009/270`.

[100] M. Turk and A. Pentland. Eigenfaces for Recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, Jan. 1991.

[101] J. Vaidya and C. Clifton. Leveraging the "Multi" in Secure Multi-Party Computation. In *ACM Workshop on Privacy in the Electronic Society*, 2003.

[102] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Theory and Practice of Software Development*, 1997.

[103] A. Waksman. A permutation network. *J. ACM*, 15:159–163, 1968.

[104] I.-C. Wang, C.-H. Shen, T.-S. Hsu, C.-C. Liao, D.-W. Wang, and J. Zhan. Towards Empirical Aspects of Secure Scalar Product. In *International Conference on Information Security and Assurance*, 2008.

[105] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC Implementation of the AES SBoxes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, 2002.

[106] H. Xu, R. Veldhuis, A. Bazen, T. Kevenaar, T. Akkermans, and B. Gokberk. Fingerprint Verification using Spectral Minutiae Representations. *IEEE Transactions on Information Forensics and Security*, 2009.

[107] A. C. Yao. How to Generate and Exchange Secrets. In *Symposium on Foundations of Computer Science*, 1986.

[108] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure Program Partitioning. *ACM Transactions on Computing Systems (TOCS)*, 20(3):283–328, 2002.

[109] J. Zhan, L. Chang, and S. Matwin. Privacy preserving k-nearest neighbor classification. *International Journal of Network Security*, 2005.

[110] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy*, 2003.