D13.1
A Set of New Protocols

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1st November, 2013
Duration:	36 months
Programme:	FP7/2007-2013
Deliverable Type:	Report
Reference Number:	ICT-609611 / D13.1 / 1.0
Activity and WP:	Activity 1 / WP13
Due Date:	October 2015 - M24
Actual Submission Date:	3rd November, 2015
Responsible Organisation:	BIU
Editor:	Benny Pinkas
Dissemination Level:	PU
Revision:	1.0
Abstract:	This document describes a set of new secure computation protocols that were designed by the partners of the PRACTICE project. These protocols were motivated by the work of deliverable D11.2, which identified shortcomings in the state-of-the-art in secure computation, mostly in terms of the scalability of existing solutions. The new protocols were published in multiple scientific papers at top-tier academic conferences in the field of computer security.
Keywords:	secure multi-party comptutation.

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 609611.

Editor

Benny Pinkas (BIU)

Contributors (ordered according to beneficiary numbers)

Florian Kerschbaum (SAP)
Florian Hahn (SAP)
Thomas Schneider (TUDA)
Michael Zohner (TUDA)
Pille Pullonen (CYBER)
Claudio Orlandi (AU)

Executive Summary

This report describes a new set of secure multi-party computation protocols that were designed by the members of the PRACTICE project. The main goal of these protocols is to enhance the performance and scalability of the available secure multi-party computation solutions, in order to address the needs of the field, which were described in Deliverable D11.2 of this project. The results that are presented in this report have been published in multiple research papers at top-tier conferences.

The first chapter of this report is an introduction. The second chapter describes new methods for generic multi-party computation (generic meaning that these methods can be used for computing arbitrary functions). The third chapter describes improvements to different tools and primitives that are used in secure computation. These improvements affect the performance of each secure computation protocol that will use these tools. The fourth chapter of this report describes new protocols for order preserving encryption, that is a crucial tool for storing encrypted databases and then performing queries on the encrypted data. The fifth and final chapter describes new methods for computing private set intersection, which is a secure protocol which, rather then being generic, solves a specific problem of high interest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main task of WP13 is the specification and design of new protocols, which are intended to improve the state-of-the-art in secure multi-party computation, in directions that are most relevant to secure computation in the cloud. Deliverable D11.2 of WP11 (An evaluation of current secure computation protocols) identified two main issues where current protocols are lacking: (1) the scalability of protocols for generic secure computation, and in particular protocols that are secure against malicious adversaries; (2) there are specific tasks, specifically private set intersection (PSI), where generic protocols are considerably less efficient (perhaps by orders of magnitude) than is required.
The work that is described in this deliverable was carried out by many members of WP13. The work mostly focused on addressing the two issues that were highlighted by deliverable D11.2. Most of the results that were achieved improved the performance and scalability of protocols for generic secure computation. Other results dramatically improved the state-of-the-art protocols for specific tasks, particularly for private set intersection (PSI), and for order preserving encryption, which is an essential tool for secure database queries.
The results that are described in this deliverable were published in the most prestigious conferences in security, as is detailed in Section 1.2.

1.1 Contents

Chapter 2 describes new methods for generic multi-party computation (generic meaning that these methods can be used for computing arbitrary functions). These new methods has considerable improved efficiency compared to the former state of the art. Section 2.1 describes a version of Yao's secure computation protocol, which is currently the basis for most secure computation solutions. The new protocol makes use of a new garbling method, which is based on standard assumptions (whereas previous state-of-the-art garbling methods depended on less established cryptographic assumptions). Section 2.2 describes the first constant round secure multi-party computation protocol that is secure against malicious adversaries and has an efficient concrete overhead. Section 2.3 describes a framework for efficient mixed-protocol two-party secure computation: secure computation protocols typically use either arithmetic circuits, where the primitive operations are addition and multiplication, or boolean circuits, where the primitive operations are XOR and AND. Each of these protocol types is better at computing different types of functions. The new framework is the first to enable easy and efficient computation which combines protocols of both types.
Chapter 3 describes improvements to different tools and primitives that are used in secure computation. These improvements affect the performance of each secure computation protocol that

will use these tools. Section 3.1 describes a new and extremely simple oblivious transfer protocol. Section 3.2 describes the first efficient oblivious transfer extension protocol that is secure against malicious adversaries. Section 3.3 describes an efficient implementation of protocols of the GMW family using hardware tokens. Section 3.4 described a new protocol stack for secure unsigned arithmetic computation for two parties, supporting more basic operations than just addition and multiplication, and thus supporting more efficient implementations. Two-Party Unsigned Arithmetic Based on Additive Secret Sharing Section 3.5 describes how to use garbled circuits for running very efficient zero-knowledge proofs of non-arithmetic statements.
Chapter 4 describes two new protocols for order preserving encryption, that is a crucial tool for storing encrypted databases and then performing queries on the encrypted data. The first protocol, in Section 4.1 improves the performance of the currently available order preserving encryption protocols. The second protocol, in Section 4.2, achieves a strictly stronger notion of security than any other order-preserving encryption scheme.
Chapter 5 describes improved new methods for private set intersection, that are based on using oblivious transfer extension and advanced hashing schemes. The chapter describes detailed experiments showing a performance improvement of more than an order of magnitude.

1.2 Publications

The results described in this document were published in the following publications, which were published in the top academic security conferences:

- Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens. Daniel Demmler, Thomas Schneider, and Michael Zohner. Usenix Security 2014.
- Optimal Average-Complexity Ideal-Security Order-Preserving Encryption. Florian Kerschbaum, and Axel Schroepfer. ACM Computer and Communication Security 2014.
- Frequency-Hiding Order-Preserving Encryption. Florian Kerschbaum, ACM Computer and Communication Security 2014.
- ABY – a Framework for Efficient Mixed-Protocol Secure Two-Party Computation. Daniel Demmler, Thomas Schneider and Michael Zohner. NDSS 2015.
- More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. Eurocrypt 2015.
- The Simplest Protocol for Oblivious Transfer. Tung Chou and Claudio Orlandi. Latincrypt 2015.
- Efficient Constant Round Multi-Party Computation Combining BMR and SPDZ. Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Crypto 2015.
- Fast Garbling of Circuits Under Standard Assumptions. Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. ACM Computer and Communication Security 2015.

Chapter 2

Improved Secure Computation Protocols

2.1 Fast Garbling of Circuits Under Standard Assumptions

A highly important tool in the design of two-party protocols is Yao's garbled circuit construction [59], and multiple optimizations on this primitive have led to performance improvements of orders of magnitude over the last years. However, many of these improvements come at the price of making very strong assumptions on the underlying cryptographic primitives being used. The justification behind making these strong assumptions has been that otherwise it is not possible to achieve fast garbling and thus fast secure computation. This work takes a step back and examines whether it is really the case that such strong assumptions are needed. It provides new methods for garbling that are secure solely under the well established assumption that the primitive used (e.g., AES) is a pseudorandom function. The results show that in many cases, the penalty incurred is not significant, and so a more conservative approach to the assumptions being used can be adopted.

Many of the optimizations to garbled circuits come at the price of assuming strong assumptions on the security of the cryptographic primitives being used. For example, the free-XOR technique requires assuming circular security as well as a type of correlation robustness [17], the usage of fixed-key AES requires assuming that AES with a fixed key behaves like a public random permutation [10], reductions in the number of encryption operations from 2 to 1 per entry in the garbled gate requires correlation robustness (when a hash function is used) and a related-key assumption (when AES is used).

Typically, the use of such non-standard cryptographic assumptions is accepted only where absolutely necessary. However, in practice, solid cryptographic engineering practices dictate a more conservative approach to assumptions. New types of elliptic curve groups are not adopted quickly, people shy away from non-standard use of block ciphers, and more. This is based on sound principles, and on the understanding that deployed solutions are very hard to change if vulnerabilities are discovered. In the field of secure computation, the willingness to take any assumption that enables a faster implementation stands in stark contrast to standard cryptographic practice. This work proposes to pause, take a step back, and ask the question how much do non-standard assumptions really cost us and are they justified.

2.1.1 The Results

This work constructed fast garbling methods solely under the assumption that AES behaves like a pseudorandom function. In particular, it does not use fixed-key AES, and uses two AES encryptions per entry in the garbled gates (since using just one encryption requires some sort of related-key security assumption). In addition, it does not use the free-XOR optimiaion (since this requires a non-standard circularity assumption). In brief, the following improvements are presented:

- **Fast AES-NI without fixing the key:** AES-NI is a hardware instruction supported on modern Intel chips, and implementing very efficient AES encryption. AES-NI can greatly benefit from pipelining the blocks that need to be encrypted. In this new work it is shown that, in addition to pipelining encryptions, it is also possible to pipeline the key schedule of AES-NI, in order to achieve very fast garbling times without using a fixed key or any other non-standard AES variant. Namely, the key schedule processing of different keys can be pipelined together, so that the amortized effect of key scheduling on Yao garbling is greatly reduced. Experiments (described below) show that this and other optimizations of AES operations have become so fast that the benefits of using fixed-key AES are almost insignificant. Thus, in contrast to current popular belief, in most cases fixed-key AES is ***not*** necessary for achieving extremely fast garbling.

- **Low-communication XOR gates:** Over the past years, it has become apparent that in secure protocols, communication is far more problematic than computation. The free-XOR technique is so attractive exactly because it requires no computation but also *no communication* for XOR gates. The paper provides a new garbling method for XOR gates that requires storing only a single ciphertext per XOR gate; the technique is inspired by the work of [45]. The computational cost is 3 AES computations for garbling the gate, and 1-2 AES computations for evaluating it.

- **Fast 4-2 row reduction:** Since the free-XOR technique is no longer used, it is possible to use 4-2 row reduction (GRR) on the non-XOR gates. However, the 4-2 row reduction method of [53] that uses polynomial interpolation is rather complex to implement (requiring finite field operations and precomputation of special constants to make it fast). In addition, even working in $GF(2^n)$ Galois fields and using the PCLMULQDQ Intel instruction, the cost is still approximately half an AES computation. The paper presents a new method for 4-2 row reduction that uses a few XOR operations only, and is trivial to implement.

The paper described implementations of these optimizations and compared them to the well known JustGarble library which is based on non-standard assumptions [10]. There is no doubt that the cost of garbling and evaluation is higher using the new and safer methods, since they have to run AES key schedules, and we pay for computing XOR gates. However, within protocol executions, the difference in performance is insignificant. This is demonstrated this by running Yao's protocol for semi-honest adversaries which has nothing but oblivious transfer (for which the fast OT extensions of [3] are used), garbled-circuit evaluation and computation, and communication.

Patent-free garbled circuits. Another considerable advantage of using the new method for computing XOR gates with low communication is that it does not rely on the free XOR

technique and thus is not patented. Since patents in cryptography are typically an obstacle to adoption, the search for efficient garbling techniques that are not patented is of great importance.

2.1.2 Experimental Results and Discussion

This work presented four tools that can optimize the performance of garbled circuits without relying on any additional cryptographic assumption beyond the existence of pseudorandom functions: (1) *pipelined garbling*; (2) *pipelined key-scheduling*; (3) *XOR gates with one ciphertext and three encryptions*; and (4) *improved 4-2 GRR for AND gates*. It then conducted experiments evaluating the performance of these methods – together and separately – and compared their performance to that of other garbling methods.
Table 2.1 shows the time it takes to run the *full Yao semi-honest protocol* [59] on three different circuits of interest: AES, SHA-256 and Min-Cut 250,000. The circuits have 6,800, 90,825 and 999,960 AND gates, respectively, and 25,124, 42,029 and 2,524,920 XOR gates, respectively. The number of input bits for which OTs are performed are 128, 256 and 250,000, respectively. The implementation of the semi-honest protocol of Yao utilizes the highly optimized OT extension protocol of [3].

Assumption		**Scheme**	AES		SHA-256		Min-Cut
			VA-VA	VA-IRE	VA-VA	VA-IRE	VA-VA
PRF	1	Pipe-garbl; XOR-3; AND-3 (naïve)	20	203	68	303	1947
	2	Pipe-garbl+KS; XOR-1; AND-3	16	200	54	236	1195
	3	**Pipe-garbl+KS; XOR-1; AND-2**	**16**	**200**	**50**	**229**	**1047**
Circularity	4	Pipe-garbl; free-XOR; AND-3	16	198	45	222	753
	5	Pipe-garbl+KS; free-XOR; AND-3	16	198	36	221	701
	6	**Pipe-garbl+KS; free-XOR; HalfGates**	**16**	**196**	**27**	**206**	**546**
Public Random Permutation	7	Fixed-key; free-XOR; AND-3	16	196	27	214	596
	8	**Fixed-key; free-XOR; Halfgates**	**16**	**195**	**20**	**199**	**460**

Table 2.1: Summary of experimental results (times are for a full semi-honest execution in milliseconds). The first row is for naïve garbling. Rows 2,3,5 and 6 are based on our improvements. The rows marked in boldface highlight the best schemes under each set of assumptions.

Assumption		**Scheme**	**AES**	**SHA-256**	**Min-Cut**
PRF	1	Pipe-garbl; XOR-3; AND-3 (naïve)	95,772	398,562	10,574,640
	2	Pipe-garbl+KS; XOR-1; AND-3	45,524	314,504	5,524,800
	3	**Pipe-garbl+KS; XOR-1; AND-2**	**38,724**	**223,679**	**4,524,840**
Related key	4	Monotone	**24,262**	300,250	3,983,114
	5	SafeMon1	29,689	217,881	4,196,607
	6	SafeMon2	37,989	**203,266**	**3,771,621**
Circularity	7	Pipe-garbl; free-XOR; AND-3	20,400	272,475	2,999,880
	8	Pipe-garbl+KS; free-XOR; AND-3	20,400	272,475	2,999,880
	9	**Pipe-garbl+KS; free-XOR; HalfGates**	**13,600**	**181,650**	**1,999,920**
Public Random Permutation	10	Fixed-key; free-XOR; AND-3	20,400	272,475	2,999,880
	11	**Fixed-key; free-XOR; Halfgates**	**13,600**	**181,650**	**1,999,920**

Table 2.2: Summary of garbled-circuit size in number of ciphertexts, according to scheme. The schemes are as in Table 2.1. (Note that for the related-key schemes, there is no single method that is always best.)

The experiments examined eight different schemes, described using the following notation: [**pipe-garble**] for the pipelined garbling method; [**pipe-garble+KS**] for pipelined garbling

and pipelined key-scheduling; [**fixed-key**] where all PRF evaluations were performed using the fixed-key technique described in [10]; [**XOR-3**] where XOR gates were garbled using a simple 4-3 GRR method; [**XOR-1**] where XOR gates were garbled using the new method of garbling with one ciphertext; [**free-XOR**] where the free-XOR technique was used; [**AND-3**] where AND gates were garbled using simple 4-3 GRR; [**AND-2**] where the new 4-2 GRR method was used to garble AND gates; and finally, [**AND-HalfGates**] where the "half-gates" technique of [62] was used to garble AND gates. Note that the half-gates method is only used in conjunction with free-XOR since this is a requirement.

The first scheme in Table 2.1 is the most "naïve", where a simple 4-3 GRR was used for both AND and XOR gates and the garbling was pipelined but not the key-scheduling. In contrast, the last scheme is the most efficient as it uses fast fixed-key encryption and the half-gates approach to achieve two ciphertexts per AND gates and none for XOR gates. However, this scheme is based on the strongest assumption, that fixed-key AES behaves like a public random permutation. The third scheme in the table uses all our optimizations together, and thus it is the most efficient scheme that is based on a standard PRF assumption. The sixth scheme in the table shows the best that can be achieved while assuming circularity and related key security, but without resorting to a public random permutation.

The experiments were performed on Amazon's c4.8xlarge compute-optimized machines (with Intel Xeon E5-2666 v3 Haswell processors) running Windows. The measurements include the time it takes to garble the circuit, send it to the evaluator and compute the output. Since communication is also involved, this measures improvements both in the encryption technique and in the size of circuit. Each scheme was tested on the three circuits in two different settings: the *Virginia-Virginia* (VA-VA) setting where the two parties running the protocol are located at the same data center, and the *Virginia-Ireland* (VA-IRE) setting where the physical distance between the parties is large. Each number in the table is an average of 20 executions of the indicated specific scenario.

The table rows marked in boldface highlight the best schemes under each set of assumptions. Looking at the results, the following observations can be derived:

- **Best efficiency:** As predicted, the fixed key + half-gates implementation (8) is the fastest and most efficient in all scenarios. (This seems trivial, but when using fixed-key AES, the eval procedure at AND gates requires one more encryption than in a simple 4-3 GRR. Thus, this confirms the hypothesis that the communication saved is far more significant than an additional encryption, that is anyway pipelined.)

- **Small circuits:** In small circuits (e.g., AES) the running time is almost identical in all schemes and in both communication settings. In particular, using the new optimizations (3) yields the same performance result as that of the most efficient scheme (8), in both the VA-VA and VA-IRE settings. This is due to the fact that in small circuits, running the OT protocol is the bottleneck of the protocol (even if optimized OT-extension [3] is used). This means that for small circuits there is no reason to rely on a non-standard cryptographic assumption.

- **Medium circuits:** In the larger SHA-256 circuit, where the majority of the gates are AND gates, there was a difference between the results in the two communication settings. In the VA-VA setting the best scheme based on PRF alone (3) has performance that is closer to that of the naïve scheme (1) than to that of the schemes based on the circularity or the public random permutation assumptions (schemes 6 and 8). In contrast, in the VA-IRE setting the PRF based scheme performs close to schemes 6 and 8. This is explained

by observing that when the parties are closely located, communication is less dominant and garbling becomes a bigger factor. Thus, garbling XOR gates for free improves the performance of the protocol. In contrast, when the parties are far from each other, communication becomes the bottleneck, thus the PRF based scheme (3) yields a significant improvement compared to the naïve case (1) and its performance is not much worse than that of the best fixed-key based scheme (and since there are fewer XOR gates, the overhead of an additional ciphertext per gate is reasonable).

- **Large circuits:** In the large Min-Cut circuit, the run time of our best PRF based scheme (3) is closer to the best result (8) than to the naïve result (1). This is explained by the fact that the circuit is very large and so bandwidth is very significant. This is especially true since the majority of gates are XOR gates, and so the reduction from 3 ciphertexts to 1 ciphertext per XOR gate has a big influence. Observe that schemes (6) and (8) have the same bandwidth; the difference in cost is therefore due to the additional cost of the AES key schedules and encryptions. Note, however, that despite the fact that there are 1,000,000 AND gates, the difference between the running-times is 15%, which is not negligible but also not overwhelming.

- **Removing the public random permutation assumption:** Comparing scheme (8), which is the most efficient, to scheme (6) which is the most efficient scheme that does not depend on the public random permutation assumption, shows that in all scenarios removing the fixed-key technique causes only a minor increase in running time.

It can be concluded that strengthening security by removing the public random permutation assumption does not noticeably affect the performance of the protocol. Thus, in many cases, two-party secure computation protocols does not need to use the fixed-key method. Further security strengthening by not depending on a circularity assumption (i.e., "paying" for XOR gates) does come with a cost. Yet, in scenarios where garbling time is not the bottleneck (e.g.,, small circuits, large inputs, communication constraints), one should consider using a more conservative approach as suggested in this work. In any case, the new ideas in this work should encourage future research on achieving faster and more efficient secure two-party computation based on standard cryptographic assumptions.

2.2 Efficient Constant Round Multi-Party Computation Combining the BMR and SPDZ Protocols

Recently, there has been much interest in the problem of concretely efficient secure MPC, where "concretely efficient" refers to protocols that are sufficiently efficient to be implemented in practice. There now exist extremely fast protocols that can be used in practice. In general, there are two approaches that have been followed; the first uses Yao's garbled circuits [61] and the second utilizes interaction for every gate like the GMW protocol [28].
There are extremely efficient variants of Yao's protocol for the two party case that are secure against malicious adversaries (e.g., [46, 47]). These protocols run in a constant number of rounds and therefore remain fast over slow networks. The BMR protocol [9] is a variant of Yao's protocol that runs in a multi-party setting with more than two parties. This protocol works by the parties jointly constructing a garbled circuit (possibly in an offline phase), and then later computing it (possibly in an online phase). However, in the case of malicious adversaries this protocol suffers from two main drawbacks: (1) Security is only guaranteed if at most

a *minority* of the parties are corrupt; (2) The protocol uses generic protocols secure against malicious adversaries (say, the GMW protocol) that evaluate the pseudorandom generator used in the BMR protocol. This non black-box construction results in an extremely high overhead. The TinyOT and SPDZ protocols [50, 22] follow the GMW paradigm, and have offline and online phases. Both of these protocols overcome the issues of the BMR protocol in that they are secure against any number of corrupt parties, make only black-box usage of cryptographic primitives, and have very fast online phases that require only very simple (information theoretic) operations. In the case of multi-party computation with more than two parties, these protocols are currently the *only practical* approach known. However, since they follow the GMW paradigm, their online phase requires a communication round for every multiplication gate. This results in a large amount of interaction and high latency, especially over slow networks. To sum up, there is no known concretely efficient constant-round protocol for the multi-party case (with the exception of [18] that considers the specific three-party case only). This new work introduces the first protocol with these properties.

Contribution This work provides the first *concretely efficient constant-round* protocol for the general *multi-party* case, with security in the presence of malicious adversaries. The basic idea behind the construction is to use an efficient non-constant round protocol – with security for malicious adversaries – to compute the gate tables of the BMR garbled circuit (and since the computation of these tables is of constant depth, this step is constant round). A crucial observation, resulting in a great performance improvement, shows that in the offline stage it is *not required to verify the correctness* of the computations of the different tables. Rather, validation of the correctness is an immediate by product of the online computation phase, and therefore does not add any overhead to the computation. Although our basic generic protocol can be instantiated with any non-constant round MPC protocol, we provide an optimized version that utilizes specific features of the SPDZ protocol [22].
In the new general construction, the new constant-round MPC protocol consists of two phases. In the first (offline) phase, the parties securely compute *random shares* of the BMR garbled circuit. If this is done naively, then the result is highly inefficient since part of the computation involves computing a pseudorandom generator or pseudorandom function multiple times for every gate. By modifying the original BMR garbled circuit, it was shown that it is possible to actually compute the circuit very efficiently. Specifically, each party locally computes the pseudorandom function as needed for every gate (the construction uses a pseudorandom function rather than a pseudorandom generator), and uses the results as input to the secure computation. The security proof shows that if a party cheats and inputs incorrect values then no harm is done, since it can only cause the honest parties to abort (which is anyway possible when there is no honest majority). Next, in the online phase, all that the parties need to do is reconstruct the single garbled circuit, exchange garbled values on the input wires and locally compute the garbled circuit. The online phase is therefore very fast.
In a concrete instantiation of the protocol, based on using the SPDZ protocol [22] in the offline phase, there are actually three separate phases, with each being faster than the previous. The first two phases can be run offline, and the last phase is run online after the inputs become known.

- The first (slow) phase depends only on an upper bound on the number of wires and the number of gates in the function to be evaluated. This phase uses Somewhat Homomorphic Encryption (SHE) and is equivalent to the offline phase of the SPDZ protocol.
- The second phase depends on the function to be evaluated but not the function inputs;

No. Parties	Beaver Triple	RandomBit	Random	Input
2	0.4	0.4	0.3	0.3
3	0.6	0.5	0.4	0.4
4	0.9	1.2	0.9	0.9

Table 2.3: SPDZ offline generation times in milliseconds per operation

in our proposed instantiation this mainly involves information theoretic primitives and is equivalent to the online phase of the SPDZ protocol.

- In the third phase the parties provide their input and evaluate the function; this phase just involves exchanging shares of the circuit and garbled values on the input wire and locally computing the BMR garbled circuit.

It should be noted that the protocol is constant round *in all phases* since the depth of the circuit required to compute the BMR garbled circuit is constant. In addition, the computational cost of preparing the BMR garbled circuit is not much more than the cost of using SPDZ itself to compute the functionality directly. However, the key advantage is that online time of the new protocol is extraordinarily fast, requiring only two rounds and local computation of a single garbled circuit. *This is faster than all other existing circuit-based multi-party protocols.*

2.2.1 Expected Runtimes

The running time of the new protocol was estimated by extrapolating from known public data [22, 21]. The offline phase of the new protocol runs both the offline and online phases of the SPDZ protocol. The numbers below refer to the SPDZ offline phase, as described in [21], with covert security and a 20% probability of cheating, using finite fields of size 128-bits, to obtain the following generation times (in milli-seconds). As described in [21], comparable times are obtainable for running in the fully malicious mode (but more memory is needed).
The implementation of the SPDZ online phase, described in both [21] and [37], reports online throughputs of between 200000 and 600000 per second for multiplication, depending on the system configuration. As remarked earlier the online time of other operations is negligible and are therefore ignored.
To see what this would imply in practice consider the AES circuit described in [53]; which has become the standard benchmarking case for secure computation calculations. The basic AES circuit has around 33000 gates and a similar number of wires, including the key expansion within the circuit. Assuming the parties share a XOR sharing of the AES key, (which adds an additional $2 \cdot n \cdot 128$ gates and wires to the circuit), the parameters for the **Initialize** call to the SPDZ functionality in the preprocessing-I protocol will be

$$M \approx 429000, \quad B \approx 33000, \quad R \approx 66000 \cdot n, \quad I \approx 66000 \cdot n + 128.$$

Using the above execution times for the SPDZ protocol it is possible to estimate the time needed for the two parts of the processing step for the AES circuit. The expected execution times, in seconds, are given in the following table. These expected times, due to the methodology of our protocol, are likely to estimate both the latency and throughput amortized over many executions.

No. Parties	preprocessing-I	preprocessing-II
2	264	0.7–2.0
3	432	0.7–2.0
4	901	0.7–2.0

The execution of the online phase of the new protocol, when the parties are given their inputs and actually want to compute the function, is very efficient: all that is needed is the evaluation of a garbled circuit based on the data obtained in the offline stage. Specifically, for each gate each party needs to process two input wires, and for each wire it needs to expand n seeds to a length which is n times their original length (where n denotes the number of parties). Namely, for each gate each party needs to compute a pseudorandom function $2n^2$ times (more specifically, it needs to run $2n$ key schedulings, and use each key for n encryptions). The cost of implementing these operations for an AES circuit of 33000 gates when the pseudorandom function is computed using the AES-NI instruction set, was examined. The run times for $n = 2, 3, 4$ parties were 6.35msec, 9.88msec and 15msec, respectively, for C code compiled using the gcc compiler on a 2.9GHZ Xeon machine. The actual run time, including all non-cryptographic operations, should be higher, but of the same order.
These run-times estimates compare favourably to several other results on implementing secure computation of AES in a multiparty setting:

- In [20] an actively secure computation of AES using SPDZ took an offline time of over five minutes per AES block, with an online time of around a quarter of a second; that computation used a security parameter of 64 as opposed to the estimates of the new protocol which use a security parameter of 128.
- In [37] another experiment was shown which can achieve a latency of 50 milliseconds in the online phase for AES (but no offline times are given).
- In [50] the authors report on a two-party MPC evaluation of the AES circuit using the Tiny-OT protocol; they obtain for 80 bits of security an amortized offline time of nearly three seconds per AES block, and an amortized online time of 30 milliseconds; but the reported non-amortized latency is much worse. Furthermore, this implementation is limited to the case of *two parties*, whereas the new protocol obtains security for multiple parties.

Most importantly, all of the above experiments were carried out in a LAN setting where communication latency is very small. However, in other settings where parties are not connect by very fast connections, the effect of the number of rounds on the protocol will be extremely significant. For example, in [20], an arithmetic circuit for AES is constructed of depth 120, and this is then reduced to depth 50 using a bit decomposition technique. Note that if parties are in separate geographical locations, then this number of rounds will very quickly dominate the running time. For example, the latency on Amazon EC2 between Virginia and Ireland is 75ms. For a circuit depth of 50, and even assuming just a *single* round per level, the running-time cannot be less than 3750 milliseconds (even if computation takes *zero time*). In contrast, the online phase of the new protocol has just 2 rounds of communication and so will take in the range of 150 milliseconds. Note that even on a much faster network with latency of just 10ms, protocols with 50 rounds of communication will still be slow.

2.3 ABY: Mixed-Protocol Secure Computation [24]

In generic secure computation, the function to be evaluated is often represented as a circuit. The two most common types of circuits that are used in secure computation are *Arithmetic circuits*, where the primitive operations are addition and multiplication, and *Boolean circuits*, where the primitive operations are XOR and AND. The efficiency of secure computation protocols goes

hand-in-hand with an efficient circuit representation of the function. For instance, performing a multiplication between two ℓ-bit values in an Arithmetic circuit is very efficient but requires a circuit of size $O(\ell^2)$ for Boolean circuits.
To overcome the dependence on an efficient function representation, several works proposed to *mix* secure computation protocols based on Arithmetic and Boolean circuits. One of these works is the ABY framework [24] for which an overview is given in Figure 2.1 and which we describe in more detail next.

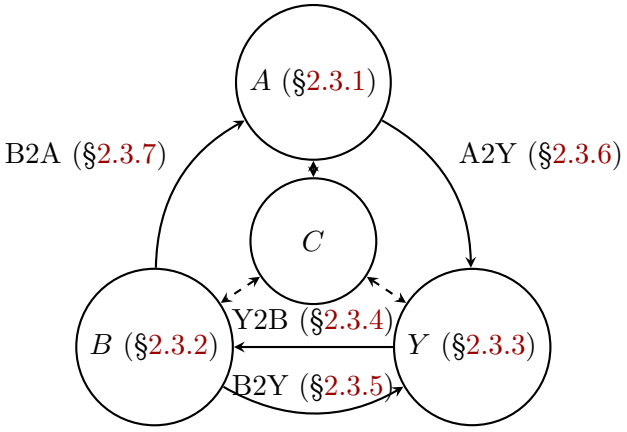


Figure 2.1: Overview of the ABY framework [24] that allows efficient conversions between **C**leartexts and three types of sharings: **A**rithmetic, **B**oolean, and **Y**ao.

Notation We denote the parties P_0 and P_1 and the symmetric security parameter as κ. We use $x[i]$ to refer to the i-bit of an element x where $x[0]$ is the least-significant bit of x. We denote a shared variable x as $\langle x \rangle^t$ where $t \in \{A, B, Y\}$ indicates the type of sharing (A for Arithmetic, B for Boolean, and Y for Yao sharing). We refer to the individual share of $\langle x \rangle^t$ that is held by party P_i as $\langle x \rangle_i^t$. We define a sharing operator $\langle x \rangle^t = \mathsf{Shr}_i^t(x)$ meaning that P_i shares its input value x with P_{1-i} and a reconstruction operator $x = \mathsf{Rec}_i^t(\langle x \rangle^t)$ meaning that P_i obtains the value of x as output. When both parties obtain the value of x, we write $\mathsf{Rec}^t(\langle x \rangle^t)$. We denote the conversion of a sharing of representation $\langle x \rangle^s$ into another representation $\langle x \rangle^d$ with $s, d \in \{A, B, Y\}$ and $s \neq d$ as $\langle x \rangle^d = s2d(\langle x \rangle^s)$, e.g., A2B converts an Arithmetic share into a Boolean share.

2.3.1 Arithmetic Sharing

For the Arithmetic sharing an ℓ-bit value x is shared additively in the ring $\mathbb{Z}_{2^\ell}$ (integers modulo 2^ℓ) as the sum of two values. The protocols described in the following are based on [6, 56, 38]. In the following, we assume all Arithmetic operations to be performed in the ring $\mathbb{Z}_{2^\ell}$, i.e., all operations are $(\text{mod } 2^\ell)$.

Sharing Semantics Arithmetic sharing is based on additively sharing private values between the parties. For an ℓ-bit Arithmetic sharing $\langle x \rangle^A$ of x we have $\langle x \rangle_0^A + \langle x \rangle_1^A \equiv x \pmod{2^\ell}$ with $\langle x \rangle_0^A, \langle x \rangle_1^A \in \mathbb{Z}_{2^\ell}$. Sharing ($\mathsf{Shr}_i^A(x)$) a value x is performed by having P_i choose $r \in_R \mathbb{Z}_{2^\ell}$, setting $\langle x \rangle_i^A = x - r$, and sending r to P_{1-i}, who sets $\langle x \rangle_{1-i}^A = r$. Reconstruction ($\mathsf{Rec}_i^A(x)$) is done by having P_{1-i} send its share $\langle x \rangle_{1-i}^A$ to P_i who computes $x = \langle x \rangle_0^A + \langle x \rangle_1^A$.

Operations Every Arithmetic circuit is a sequence of addition and multiplication gates. Addition $\langle z\rangle^A = \langle x\rangle^A + \langle y\rangle^A$ is done by having P_i locally compute $\langle z\rangle_i^A = \langle x\rangle_i^A + \langle y\rangle_i^A$. Multiplication $\langle z\rangle^A = \langle x\rangle^A \cdot \langle y\rangle^A$ is performed using a pre-computed Arithmetic multiplication triple [7] of the form $\langle c\rangle^A = \langle a\rangle^A \cdot \langle b\rangle^A$ that can be pre-computed by using either additively homomorphic encryption or OT extension as described in [24]. To multiply, P_i sets $\langle e\rangle_i^A = \langle x\rangle_i^A - \langle a\rangle_i^A$ and $\langle f\rangle_i^A = \langle y\rangle_i^A - \langle b\rangle_i^A$, both parties perform $\mathsf{Rec}^A(e)$ and $\mathsf{Rec}^A(f)$, and P_i sets $\langle z\rangle_i^A = i \cdot e \cdot f + f \cdot \langle a\rangle_i^A + e \cdot \langle b\rangle_i^A + \langle c\rangle_i^A$.

2.3.2 Boolean Sharing

In Boolean sharing we evaluate Boolean circuits using the GMW protocol [28].

Sharing Semantics Boolean sharing uses an XOR-based secret sharing scheme. To simplify presentation, we assume single bit values; for ℓ-bit values each operation is performed ℓ times in parallel. A Boolean share $\langle x\rangle^B$ of a bit x is shared between the two parties, such that $\langle x\rangle_0^B \oplus \langle x\rangle_1^B = x$ with $\langle x\rangle_0^B, \langle x\rangle_1^B \in \mathbb{Z}_2$. Sharing ($\mathsf{Shr}_i^B(x)$) is done by having P_i choose $r \in_R \{0,1\}$, compute $\langle x\rangle_i^B = x \oplus r$, and send r to P_{1-i} who sets $\langle x\rangle_{1-i}^B = r$. Reconstruction ($\mathsf{Rec}_i^B(x)$) is done by having P_{1-i} send its share $\langle x\rangle_{1-i}^B$ to P_i who computes $x = \langle x\rangle_0^B \oplus \langle x\rangle_1^B$.

Operations Every efficiently computable function can be expressed as a Boolean circuit consisting of XOR and AND gates. The XOR operation $\langle z\rangle^B = \langle x\rangle^B \oplus \langle y\rangle^B$ is computed by having P_i locally compute $\langle z\rangle_i^B = \langle x\rangle_i^B \oplus \langle y\rangle_i^B$. The AND operation $\langle z\rangle^B = \langle x\rangle^B \wedge \langle y\rangle^B$ is evaluated using a pre-computed Boolean multiplication triple $\langle c\rangle^B = \langle a\rangle^B \wedge \langle b\rangle^B$ as follows: P_i computes $\langle e\rangle_i^B = \langle a\rangle_i^B \oplus \langle x\rangle_i^B$ and $\langle f\rangle_i^B = \langle b\rangle_i^B \oplus \langle y\rangle_i^B$, both parties perform $\mathsf{Rec}^B(e)$ and $\mathsf{Rec}^B(f)$, and P_i sets $\langle z\rangle_i^B = i \cdot e \cdot f \oplus f \cdot \langle a\rangle_i^B \oplus e \cdot \langle b\rangle_i^B \oplus \langle c\rangle_i^B$. As described in [3], a Boolean multiplication triple can be pre-computed efficiently using OT extension. For other standard functionalities (such as addition and comparison) we use the depth-optimized circuit constructions summarized in [58].

2.3.3 Yao Sharing

In Yao's garbled circuits protocol [59] a garbler (P_0) represents the function to be computed as Boolean circuit and assigns to each wire w two wire keys (k_0^w, k_1^w) with $k_0^w, k_1^w \in \{0,1\}^\kappa$. The garbler then encrypts (garbles) the Boolean circuit and sends the garbled circuit to the evaluator (P_1) who iteratively decrypts each garbled gate to obtain the output of the circuit. In the following we detail the Yao sharing assuming a garbling scheme that uses the free-XOR [44] and point-and-permute [48] optimizations. Using these techniques, the garbler randomly chooses a global κ-bit string R with $R[0] = 1$. For each wire w, the wire keys are $k_0^w \in_R \{0,1\}^\kappa$ and $k_1^w = k_0^w \oplus R$. The least significant bit $k_0^w[0]$ resp. $k_1^w[0] = 1 - k_0^w[0]$ is called permutation bit.

Sharing Semantics Intuitively, P_0 holds for each wire w the two keys k_0^w and k_1^w and P_1 holds one of these keys without knowing to which of the two cleartext values it corresponds. To simplify presentation, we assume single bit values; for ℓ-bit values each operation is performed ℓ times in parallel. A garbled circuit share $\langle x\rangle^Y$ of a value x is shared as $\langle x\rangle_0^Y = k_0$ and $\langle x\rangle_1^Y = k_x = k_0 \oplus xR$. Sharing for P_0 ($\mathsf{Shr}_0^Y(x)$) is done by having P_0 sample $\langle x\rangle_0^Y = k_0 \in_R \{0,1\}^\kappa$ and send $k_x = k_0 \oplus xR$ to P_1. Sharing for P_1 ($\mathsf{Shr}_1^Y(x)$) is done by having both parties run OT where P_0 acts as sender and inputs $(k_0, k_1 = k_0 \oplus R)$ with $k_0 \in_R \{0,1\}^\kappa$ and P_1 acts as

receiver with choice bit x and obliviously obtains $\langle x\rangle_1^Y = k_x$. Reconstruction ($\mathsf{Rec}_i^Y(x)$) is done by having P_{1-i} send its permutation bit $\pi = \langle x\rangle_{1-i}^Y[0]$ to P_i who computes $x = \pi \oplus \langle x\rangle_i^Y[0]$.

Operations Yao sharing evaluates a Boolean circuit consisting of XOR and AND gates. XOR gates $\langle z\rangle^Y = \langle x\rangle^Y \oplus \langle y\rangle^Y$ are evaluated using the free-XOR technique [44], i.e., by having: P_i locally compute $\langle z\rangle_i^Y = \langle x\rangle_i^Y \oplus \langle y\rangle_i^Y$. AND gates $\langle z\rangle^Y = \langle x\rangle^Y \wedge \langle y\rangle^Y$ are evaluated by having P_0 create a garbled table using $\mathsf{Gb}_{\langle z\rangle_0^Y}\left(\langle x\rangle_0^Y, \langle y\rangle_0^Y\right)$, where Gb is a garbling function as defined in [10]. P_0 sends the garbled table to P_1, who decrypts it using the keys $\langle x\rangle_1^Y$ and $\langle y\rangle_1^Y$. For standard functionalities we use the size-optimized circuit constructions summarized in [43].

2.3.4 Yao to Boolean Sharing (Y2B)

Converting a Yao share $\langle x\rangle^Y$ to a Boolean share $\langle x\rangle^B$ is the easiest conversion and comes essentially for free. The key insight is that the permutation bits of $\langle x\rangle_0^Y$ and $\langle x\rangle_1^Y$ already form a valid Boolean sharing of x. Thus, P_i locally sets $\langle x\rangle_i^B = Y2B(\langle x\rangle_i^Y) = \langle x\rangle_i^Y[0]$.

2.3.5 Boolean to Yao Sharing (B2Y)

Converting a Boolean share $\langle x\rangle^B$ to a Yao share $\langle x\rangle^Y$ is very similar to the Shr_1^Y operation (cf. §2.3.3): In the following we assume that x is a single bit; for ℓ-bit values, each operation is done ℓ times in parallel. Let $x_0 = \langle x\rangle_0^B$ and $x_1 = \langle x\rangle_1^B$. P_0 samples $\langle x\rangle_0^Y = k_0 \in_R \{0,1\}^\kappa$. Both parties run $1 \times OT_\kappa$ where P_0 acts as sender with inputs $(k_0 \oplus x_0 \cdot R; k_0 \oplus (1-x_0)\cdot R)$, whereas P_1 acts as receiver with choice bit x_1 and obliviously obtains $\langle x\rangle_1^Y = k_0 \oplus (x_0 \oplus x_1)\cdot R = k_x$.

2.3.6 Arithmetic to Yao Sharing (A2Y)

Converting an Arithmetic share $\langle x\rangle^A$ to a Yao share $\langle x\rangle^Y$ was outlined in [30, 38] and can be done by securely evaluating an addition circuit. More precisely, the parties secret share their Arithmetic shares $x_0 = \langle x\rangle_0^A$ and $x_1 = \langle x\rangle_1^A$ as $\langle x_0\rangle^Y = \mathsf{Shr}_0^Y(x_0)$ and $\langle x_1\rangle^Y = \mathsf{Shr}_1^Y(x_1)$ and compute $\langle x\rangle^Y = \langle x_0\rangle^Y + \langle x_1\rangle^Y$.

2.3.7 Boolean to Arithmetic Sharing (B2A)

The general idea of the B2A conversion is to perform an OT for each bit where we obliviously transfer two values that are additively correlated by a power of two. The receiver can obtain one of these values and, by summing them up, the parties obtain a valid Arithmetic share. More detailed, P_0 acts as sender and P_1 acts as receiver in the OT protocol. In the i-th OT, P_0 randomly chooses $r_i \in_R \{0,1\}^\ell$ and inputs $(s_{i,0}, s_{i,1})$ with $s_{i,0} = \left(1 - \langle x\rangle_0^B[i]\right)\cdot 2^i - r_i$ and $s_{i,1} = \langle x\rangle_0^B[i]\cdot 2^i - r_i$, whereas P_1 inputs $\langle x\rangle_1^B[i]$ as choice bit and receives $s_{\langle x\rangle_1^B[i]} = \left(\langle x\rangle_0^B[i] \oplus \langle x\rangle_1^B[i]\right)\cdot 2^i - r_i$ as output. Finally, P_0 computes $\langle x\rangle_0^A = \sum_{i=1}^\ell r_i$ and P_1 computes $\langle x\rangle_1^A = \sum_{i=1}^\ell s_{\langle x\rangle_1^B[i]} = \sum_{i=1}^\ell \left(\langle x\rangle_0^B[i] \oplus \langle x\rangle_1^B[i]\right)\cdot 2^i - \sum_{i=1}^\ell r_i = \sum_{i=1}^\ell x[i]\cdot 2^i - \sum_{i=1}^\ell r_i = x - \langle x\rangle_0^A$.

Chapter 3

Tools with Improved Efficiency

3.1 Simple and Efficient Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic primitive defined as follows: in its simplest flavour, 1-out-of-2 OT, a sender has two input messages M_0 and M_1 and a receiver has a choice bit c. At the end of the protocol the receiver is supposed to learn the message M_c and nothing else, while the sender is supposed to learn nothing. Perhaps surprisingly, this extremely simple primitive is sufficient to implement any cryptographic task [41]. OT is also one of the main building blocks in most secure-two party computation protocol such as Yao's garbled circuits, the GMW protocol etc.
Given the importance of OT, and the fact that most OT applications require a very large number of OTs, it is crucial to construct OT protocols which are at the same time efficient and secure against realistic adversaries.

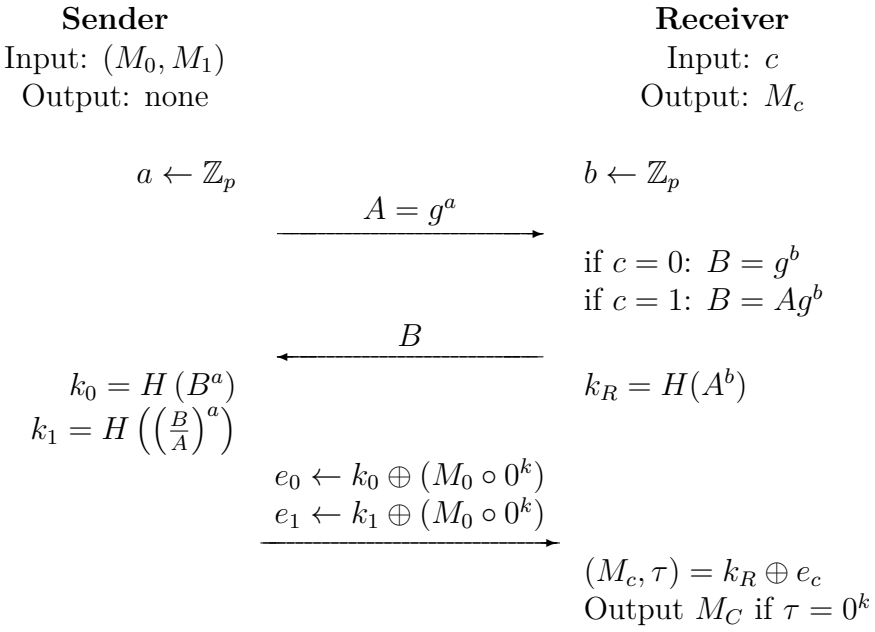


Figure 3.1: Our protocol in a nutshell

3.1.1 A Novel OT Protocol

In a recent work [19] (co-authored by AU and TUE) a novel, extremely *simple*, *efficient* and *secure* OT protocol was presented. The protocol is a simple tweak of the celebrated Diffie-Hellman (DH) key exchange protocol. Given a group $\mathbb{G}$ and a generator g, the DH protocol allows two players Alice and Bob to agree on a key as follows: Alice samples a random a, computes $A = g^a$ and sends A to Bob. Symmetrically Bob samples a random b, computes $B = g^b$ and sends B to Alice. Now both parties can compute $g^{ab} = A^b = B^a$ from which they can derive a key k. The key observation is now that Alice can also derive a different key from the value $(B/A)^a = g^{ab-a^2}$, and that Bob cannot compute this group element (assuming that the computational DH problem is hard).
We can now turn this into an OT protocol by letting Alice play the role of the sender and Bob the role of the receiver (with choice bit c) as shown in Figure 3.1. The first message (from Alice to Bob) is left unchanged (and can be reused over multiple instances of the protocol) but now Bob computes B as a function of his choice bit c: if $c = 0$ Bob computes $B = g^b$ and if $c = 1$ Bob computes $B = Ag^b$. At this point Alice derives two keys k_0, k_1 from $(B)^a$ and $(B/A)^a$ respectively. It is easy to check that Bob can derive the key k_c corresponding to his choice bit from A^b, but cannot compute the other one. This can be seen as a *random OT* i.e., an OT where the sender has no input but instead receives two random messages from the protocol, which can be used later to encrypt his inputs.
Combining the above random OT protocol with the right symmetric encryption scheme (e.g., a *robust encryption scheme*) achieves security in a strong, simulation based sense and in particular the protocol can be proven UC-secure against active and adaptive corruptions in the random oracle model.

Experimental Validation. The paper also reports on an efficient and secure implementation of the random OT protocol: the chosen group is a twisted Edwards curve that has been used by Bernstein, Duif, Lange, Schwabe and Yang for building a *high-speed high-security* signature scheme [12]. The security of the curve comes from the fact that it is birationally equivalent to Bernstein's Montgomery curve Curve25519 where ECDLP is believed to be hard.
The implementation uses the code in [12]: In order to make use of the natural parallelism in the protocol, a vectorized implementation for the Intel Sandy Bridge and Ivy Bridge microarchitectures was built. A comparison with the state of the art shows that our implementation is at least an order of magnitude faster than previous work (we compare in particular with the implementation reported by Asharov, Lindell, Schneider and Zohner in [5]). Furthermore, the code has been carefully implemented to make sure that our implementation is secure against both passive attacks (our software is *immune to timing attacks*, since the implementation is *constant-time*) and active attacks (by designing an appropriate encoding of group elements, which can be efficiently verified and computed on).

3.2 Active Secure Oblivious Transfer Extension [4]

Oblivious Transfer (OT) is a cryptographic primitive that is fundamental for secure computation. In an 1-out-of-2 OT, a sender P_S holds a pair of n-bit strings (x^0, x^1) of which a receiver P_R with choice bit r wants to obtain x^r such that P_S does not learn r and P_R gains no information about x^{1-r}.
It has been proven that OT can not be based on one-way functions alone [33] and hence computing an OT requires public-key cryptography. However, public-key cryptography is very

costly and many applications in secure computation typically require millions up to billions of OTs, so getting efficient OT is of high importance. In [8] it was shown that a small number of real *base-OTs*, that were computed using OT protocols based on expensive public-key cryptography, can be extended to an arbitrarily large number of OTs using efficient symmetric cryptography only. Due to their nature, these protocols are called *OT extension* protocols.
While the feasibility result of [8] was still costly in concrete terms, the work of [34] showed how to extend OTs at a relatively low cost. The main protocol that was introduced was secure against passive (or semi-honest) adversaries, which try to learn as much information as possible but honestly follow the protocol description. An extension for security against active (or malicious) adversaries, which can arbitrarily deviate from the protocol description, was also given but incurred a huge cost overhead (around factor 40 compared to the passively secure variant).
Several follow-up works reduced the cost for both the passively- and actively secure variants of the protocol. One of these works is [4], which reduced the cost overhead of the active secure protocol over the passive secure protocol to factor 1.4. In Protocol 1 we give the malicious OT extension protocol of [4].

3.3 Token-Aided Mobile GMW [23]

In the two-party case, the Goldreich-Micali-Wigderson (GMW) protocol [28] enables two mutually-distrusting parties $\mathcal{A}$ and $\mathcal{B}$ to securely evaluate a function which is expressed as a Boolean circuit. The GMW protocol relies heavily on OT and requires two OTs to compute a *multiplication triple* [7] that is used to evaluate an AND gate. Using OT pre-computation [7], the computation and communication intensive operations can be pre-computed in an interactive *setup phase* that is independent of the function. By pre-computing the multiplication triples, the *online phase*, where the actual function is being evaluated, becomes very efficient as it consists of only primitive operations (XOR and AND) and requires only little communication.
While it was shown that secure computation can be very efficiently performed on Desktop PCs [58], it still is impractical on resource-constrained devices such as mobile phones. However, in spite of being resource-constrained, mobile phones can be equipped with devices such as *hardware tokens* that can be used to increase the efficiency of secure computation. In the following section, we give details on our hardware token-aided GMW-based protocol for mobile phones [23]. Our goal is to minimize the *ad-hoc time* of the protocol, i.e., the time from establishing the communication channel between party $\mathcal{A}$ and party $\mathcal{B}$ until receiving the results of the secure computation. We consider the *init phase*, which can be performed by the parties before establishing a communication channel, to not be time critical, but we try to keep its computational overhead small.
An overview of our protocol is given in Figure 3.2. The general idea is to let the hardware token $\mathcal{T}$, held by $\mathcal{A}$, generate multiplication triples from two (or more) seeds in the init phase that are independent of the later communication partner (§3.3.1). In the setup phase, $\mathcal{T}$ then sends one seed to $\mathcal{A}$ and the other seed over an encrypted channel to $\mathcal{B}$ (§3.3.2). The token thereby replaces the OT protocol in the setup phase and allows pre-computing the multiplication triples independently of the communication partner. The online phase of the GMW protocol remains unchanged.

3.3.1 Multiplication Triple Pre-Computation in the Init Phase

In the original GMW protocol, $\mathcal{A}$ and $\mathcal{B}$ interactively compute their multiplication triples $(a^n_\mathcal{A}, b^n_\mathcal{A}, c^n_\mathcal{A})$ and $(a^n_\mathcal{B}, b^n_\mathcal{B}, c^n_\mathcal{B})$ where $c^n_\mathcal{A} \oplus c^n_\mathcal{B} = (a^n_\mathcal{A} \oplus a^n_\mathcal{B}) \wedge (b^n_\mathcal{A} \oplus b^n_\mathcal{B})$ in the setup phase using

PROTOCOL 1 (Active secure OT extension protocol of [4].)

- **Input of P_S:** m pairs (x_j^0, x_j^1) of n-bit strings, $1 \le j \le m$.
- **Input of P_R:** m selection bits $\mathbf{r} = (r_1, \ldots, r_m)$.
- **Common Input:** Symmetric security parameter κ and statistical security parameter ρ. It is assumed that the number of base-OTs is $\ell = \kappa + \rho$.
- **Oracles and cryptographic primitives: The parties use an ideal $\ell \times OT_\kappa$ functionality, which computes ℓ OTs on κ-bit input values, pseudorandom generator $G : \{0,1\}^\kappa \to \{0,1\}^m$, and random-oracle H.**

1. *Initial OT Phase:*
   (a) P_S initializes a random vector $\mathbf{s} = (s_1, \ldots, s_\ell) \in \{0,1\}^\ell$ and P_R chooses ℓ pairs of seeds $\mathbf{k}_i^0, \mathbf{k}_i^1$ each of size κ.
   (b) The parties invoke the $\ell \times OT_\kappa$-functionality, where P_S acts as the *receiver* with input $\mathbf{s}$ and P_R acts as the *sender* with inputs $(\mathbf{k}_i^0, \mathbf{k}_i^1)$ for every $1 \le i \le \ell$.

   For every $1 \le i \le \ell$, let $\mathbf{t}^i = G(\mathbf{k}_i^0)$. Let $T = [\mathbf{t}^1 | \ldots | \mathbf{t}^\ell]$ denote the $m \times \ell$ bit matrix where its ith column is $\mathbf{t}^i$ for $1 \le i \le \ell$. Let $\mathbf{t}_j$ denote the jth row of T for $1 \le j \le m$.
2. *OT Extension Phase (Part I):*
   (a) P_R computes $\mathbf{t}^i = G(\mathbf{k}_i^0)$ and $\mathbf{u}^i = \mathbf{t}^i \oplus G(\mathbf{k}_i^1) \oplus \mathbf{r}$, and sends $\mathbf{u}^i$ to P_S for every $1 \le i \le \ell$.
3. *Consistency Check of $\mathbf{r}$:*
   (a) For every pair $\alpha, \beta \subseteq [\ell]^2$, P_R defines the four values:

$$h_{\alpha,\beta}^{0,0} = H(G(\mathbf{k}_\alpha^0) \oplus G(\mathbf{k}_\beta^0)) \qquad h_{\alpha,\beta}^{0,1} = H(G(\mathbf{k}_\alpha^0) \oplus G(\mathbf{k}_\beta^1)) \ ,$$
$$h_{\alpha,\beta}^{1,0} = H(G(\mathbf{k}_\alpha^1) \oplus G(\mathbf{k}_\beta^0)) \qquad h_{\alpha,\beta}^{1,1} = H(G(\mathbf{k}_\alpha^1) \oplus G(\mathbf{k}_\beta^1)) \ .$$

   It then sends $\mathcal{H}_{\alpha,\beta} = (h_{\alpha,\beta}^{0,0}, h_{\alpha,\beta}^{0,1}, h_{\alpha,\beta}^{1,0}, h_{\alpha,\beta}^{1,1})$ to P_S.
   (b) For every pair $\alpha, \beta \subseteq [\ell]^2$, P_S knows $s_\alpha, s_\beta, \mathbf{k}_\alpha^{s_\alpha}, \mathbf{k}_\beta^{s_\beta}, \mathbf{u}^\alpha, \mathbf{u}^\beta$ and checks that:
      i. $h_{\alpha,\beta}^{s_\alpha, s_\beta} = H(G(\mathbf{k}_\alpha^{s_\alpha}) \oplus G(\mathbf{k}_\beta^{s_\beta}))$.
      ii. $h_{\alpha,\beta}^{\overline{s_\alpha}, \overline{s_\beta}} = H(G(\mathbf{k}_\alpha^{s_\alpha}) \oplus G(\mathbf{k}_\beta^{s_\beta}) \oplus \mathbf{u}^\alpha \oplus \mathbf{u}^\beta) \quad (= H(G(\mathbf{k}_\alpha^{\overline{s_\alpha}}) \oplus G(\mathbf{k}_\beta^{\overline{s_\beta}}) \oplus \mathbf{r}^\alpha \oplus \mathbf{r}^\beta))$.
      iii. $\mathbf{u}^\alpha \neq \mathbf{u}^\beta$.

   In case one of these checks fails, P_S aborts and outputs $\perp$.
4. *OT Extension Phase (Part II):*
   (a) For every $1 \le i \le \ell$, P_S defines $\mathbf{q}^i = (s_i \cdot \mathbf{u}^i) \oplus G(\mathbf{k}_i^{s_i})$. (Note that $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$.)
   (b) Let $Q = [\mathbf{q}^1 | \ldots | \mathbf{q}^\ell]$ denote the $m \times \ell$ bit matrix where its ith column is $\mathbf{q}^i$. Let $\mathbf{q}_j$ denote the jth row of the matrix Q. (Note that $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$ and $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$.)
   (c) P_S sends (y_j^0, y_j^1) for every $1 \le j \le m$, where:

$$y_j^0 = x_j^0 \oplus H(j, \mathbf{q}_j) \quad \text{and} \quad y_j^1 = x_j^1 \oplus H(j, \mathbf{q}_j \oplus \mathbf{s})$$

   (d) For $1 \le j \le m$, P_R computes $x_j = y_j^{r_j} \oplus H(j, \mathbf{t}_j)$.
5. **Output:** P_R outputs $(x_1^{r_1}, \ldots, x_m^{r_m})$; P_S has no output.

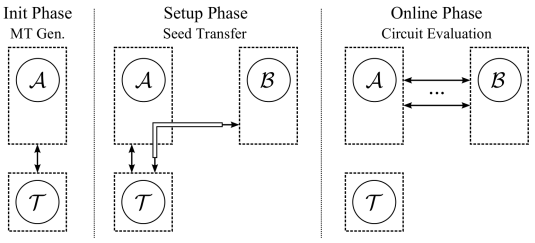
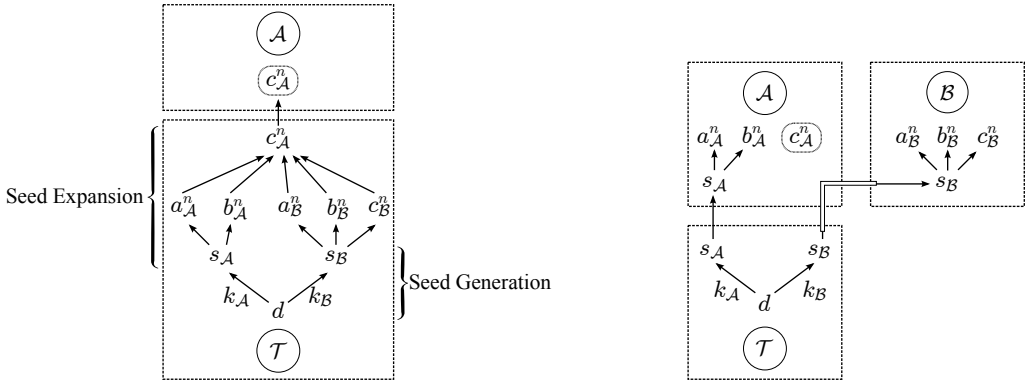


Figure 3.2: The three phases, workload distribution, and communication in our token-aided scheme.

$2n$ random OT extensions. Instead, we avoid this overhead in the setup phase and let $\mathcal{T}$ pre-compute the multiplication triples in the init phase as shown in Figure 3.3(a): $\mathcal{T}$ first *generates* random seeds and then *expands* these seeds internally into the multiplication triples and sends $c_{\mathcal{A}}^n$ to $\mathcal{A}$.



(a) Init Phase (b) Setup Phase

Figure 3.3: Multiplication triple pre-generation in the init phase between $\mathcal{A}$ and $\mathcal{T}$ (a) and seed transfer and seed expansion in the setup phase (b). $s_{\mathcal{B}}$ is sent from $\mathcal{T}$ to $\mathcal{B}$ over a secure channel.

Seed Generation In the seed generation step, $\mathcal{T}$ generates two seeds $s_{\mathcal{A}} = G_{k_{\mathcal{A}}}(d)$ and $s_{\mathcal{B}} = G_{k_{\mathcal{B}}}(d)$ using a cryptographically strong Pseudo-Random Generator (PRG) G, two master keys $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$, and a state value d, which is unique per multiplication triple sequence and can be instantiated with a counter. The two master keys $k_{\mathcal{A}}$ and $k_{\mathcal{B}}$ are constant for all multiplication triple sequences and have to be generated and stored only once. Thereby, $\mathcal{T}$ has to store only the unique state value d in its internal memory for every multiplication triple sequence. Note that the only values that will leave the internal memory of $\mathcal{T}$ are the seeds $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$ that will be sent in the setup phase to $\mathcal{A}$ and $\mathcal{B}$, respectively (cf. §3.3.2). In order to ensure that $s_{\mathcal{B}}$ is not sent out twice, we require $s_{\mathcal{A}}$ to be queried before $s_{\mathcal{B}}$ and delete the state value d as soon as $s_{\mathcal{B}}$ has been sent out over the encrypted channel.

Seed Expansion The seed expansion step computes a valid multiplication triple sequence from the seeds $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$ by computing $(a_{\mathcal{A}}^n, b_{\mathcal{A}}^n) = G_{s_{\mathcal{A}}}(d_{\mathcal{A}})$ and $(a_{\mathcal{B}}^n, b_{\mathcal{B}}^n, c_{\mathcal{B}}^n) = G_{s_{\mathcal{B}}}(d_{\mathcal{B}})$ and setting the remaining value $c_{\mathcal{A}}^n = (a_{\mathcal{A}}^n \oplus a_{\mathcal{B}}^n) \wedge (b_{\mathcal{A}}^n \oplus b_{\mathcal{B}}^n) \oplus c_{\mathcal{B}}^n$, where $d_{\mathcal{A}}$ and $d_{\mathcal{B}}$ are publicly known state values of $\mathcal{A}$ and $\mathcal{B}$, respectively. Due to the limited memory of the hardware token, the sequence $c_{\mathcal{A}}^n$ is computed block-wise such that $\mathcal{T}$ requires only a fixed amount of memory, independently of n, and each block is sent to $\mathcal{A}$, who stores it locally. Note that the

values $(a^n_{\mathcal{A}}, b^n_{\mathcal{A}}, a^n_{\mathcal{B}}, b^n_{\mathcal{B}}, c^n_{\mathcal{B}})$ do not need to be stored, since they can be expanded from $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$, respectively.

3.3.2 Seed Transfer in the Setup Phase

In the setup phase, the hardware token sends the seeds $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$ to $\mathcal{A}$ and $\mathcal{B}$, respectively, and the parties generate their multiplication triples as depicted in Figure 3.3(b). $\mathcal{A}$ obtains his seed $s_{\mathcal{A}}$ directly from $\mathcal{T}$ and can read the sequence $c^n_{\mathcal{A}}$, which was obtained in the init phase, from its internal flash storage. $\mathcal{B}$'s seed $s_{\mathcal{B}}$, on the other hand, cannot be sent in plaintext from $\mathcal{T}$ to $\mathcal{B}$, as the communication between the token and $\mathcal{B}$ is relayed over $\mathcal{A}$, which would allow $\mathcal{A}$ to intercept $s_{\mathcal{B}}$ and thus break the security of the scheme. We therefore require the communication between $\mathcal{B}$ and $\mathcal{T}$ to be encrypted and $\mathcal{T}$ to authenticate itself to $\mathcal{B}$ with a certificate, signed by a trusted-third party. To establish this communication channel, we use *TLS* [32].

3.4 Two-Party Unsigned Arithmetic Based on Additive Secret Sharing

3.4.1 Introduction

This chapter proposes a protocol stack for secure unsigned arithmetic computation for two parties. Although in theory addition and multiplication suffice for all computation, it is often more efficient to use specialized protocols also for other operations. We describe common arithmetic protocols like addition, multiplication, integer division, exponentiation and comparisons. In addition, we include bit level operations like shifts and rotations. Some operations have protocols for different flavours where some inputs can be public instead of private to get more efficiency. For example, division has two versions with either public (PubDiv) or private divisor (PrivDiv).

We focus on computations in rings Z_{2^k} for some integer $k > 0$ which correspond to the k-bit unsigned integer data types. An additive secret sharing of element x is denoted as $[\![x]\!]$. Sharing $[\![x]\!]$ is made up of two shares $[\![x]\!]_1$ and $[\![x]\!]_2$ where $x = [\![x]\!]_1 + [\![x]\!]_2$. These shares are distributed so that party $\mathcal{CP}_i$ for $i \in \{1, 2\}$ has share $[\![x]\!]_i$ and no knowledge about the other share. Usually, the ring where the sharing is computed is understandable from the context, but it is also possible to denote it as $[\![x]\!]_{\bmod 2^k}$ for $x \in \mathbb{Z}_{2^k}$ where $[\![x]\!]_1, [\![x]\!]_2 \in \mathbb{Z}_{2^k}$. Furthermore, it is possible to access single bits of individual shares or extract them from the shared elements. For that, $x[k]$ denotes the k'th bit of the value x, where $k = 1$ means the least significant bit. In case of the share of the first party, the k'th bit would be denoted as $[\![x]\!]_1[k]$. However, in case the shared bit is extracted from the shared value the result is denoted as $[\![x[k]]\!]_{\bmod 2}$.

We assume that at most one of the parties is passively corrupted and ensure the security of the computations in this case. Current protocols rely on the security proof framework of passively secure protocols from [15]. A significant step when using this framework is to determine secure protocols that are a suitable finishing step of the composition. Two straightforward candidates for this role are resharing and declassify. Declassify is more meaningful in the two-party setting as it is a natural finishing step of any protocol. Clearly, two-party declassify is a secure protocol as knowing $[\![x]\!]_1$ and x the second share is uniquely fixed as $x - [\![x]\!]_1$ and can be perfectly simulated. The rest of the protocols in the computation are allowed to be input private.

3.4.2 Overview of the Protocol Stack

The goal of this work is to design efficient protocols for all kinds of unsigned integer arithmetic, for example, addition, multiplication, division, exponentiation and bit shifts. In addition to well-known operations, some sub-protocols are required to build the desired functionalities. Table 3.1 lists the proposed operations as well as dependencies between different protocols. Note that only direct dependencies are listed and the prerequisites of the sub-protocols are not recursively added. Most of these protocols operate on additive sharing of unsigned integers, but also refer to the bitwise sharing protocols that work only in $\mathbb{Z}_2$. It is further work to determine if replacing some of the the binary protocols with garbled circuits similarly to ABY framework [25] would be useful.

Name	Description	Dependencies
Add	Addition (also exclusive-or)	
Neg	Negation	
Sub	Subtraction	Add, Neg
Classify	Sharing secret input	
Declassify	Public output from shares	
PubMult	Multiplication with a constant	Add
Mult	Multiplication (also conjunction)	Add, Declassify, Precomputation
PrefixOR	Cumulative OR of prefixes	Add, Mult
MSNZB	Most significant non-zero bit	Add, PrefixOR
Overflow	1, if $[\![x]\!]_1 + [\![x]\!]_2 \geq 2^k$	Add, Classify, Mult, MSNZB
ShiftL	Left shift by a public shifter	PubMult
SiftR	Right shift by a public shifter	Add, Extend, Overflow, PubMult
BitRotate	Bit rotation right by public value	Add, ShiftL, ShiftR
BitConj	AND of a bit-vector	Mult
EqZero	1 iff shared input is equal to 0	BitConj, Sub
Eq	Equality of two shared elements	EqZero, Sub
BitExt	Extension of bitwise sharing to additive sharing	Add, Mult, PubMult
Extend	Conversion to a larger domain	BitExt, Overflow
Trunc	Conversion to a smaller domain	
PubDiv	Division with a public divisor	Add, Extend, Overflow, Mult, PubMult, Trunc
Remainder	Remainder for a public modulus	PubDiv, PubMult, Sub
i'th-Bit	Extraction of a single bit	Add, Overflow
LT	Less than of two shared inputs	Add, i'th-Bit, Mult, Sub
LTE	Less than or equal	Add, Eq, LT, Mult, Neg
BitExtr	Bit extraction	Add, Classify, Mult
PrivDiv	Division with a private divisor	Add, BitExtr, Extend, Overflow, Mult, MSNZB, PubMult, ShiftR, Sub
ChVector	Characteristic vector $[\![v_0]\!]$ $\ldots, [\![v_\ell]\!]$ of $[\![x]\!]$ where $v_i = 1$ iff $x = i$, otherwise $v_i = 0$	Declassify, Sub, Precomputed random element $[\![r]\!]$ and ChVector($[\![r]\!]$)

PubExp	Exponentiation with a public base and shared exponent	Add, ChVector, PubMult
PrivShiftL	Left shift by a private shifter	Mult, PubExp
PrivShiftR	Right shift by a private shifter	Add, BitExtr, ChVector, Extend, LT, Mult

Table 3.1: Protocols dependencies for additive secret sharing based unsigned integer computation.

Most the protocols in Table 3.1 can be achieved quite easily. For example, addition (Add) is defined by the properties of the additive sharing scheme and can be computed locally. The same holds for subtraction (Sub) and negation (Neg) that can also be evaluated locally by each party. For classifying a secret (Classify), a party creates and distributes the shares. For declassifying (Declassify), the shares are sent to the party who requires the result. For multiplication (Mult), the precomputed multiplication with Beaver triples [7] is used, however, currently the precomputation method is not specified. For example, ideas from ABY framework [25] are applicable in our setting.
Out of the building block protocols PrefixOR and MSNZB can be taken directly from [16]. Some protocols such as Overflow, ShiftR and EqZero can be easily adapted from their three-party counterparts in [16] as the three-party protocols need an extra step to reshare the value to two parties. This allows the three-party versions to basically revert to two-party protocol for easier handling of possible overflows and directly gives rise to full two-party protocols. The advanced PrivDiv protocol from [16] that performs integer division of secret shared inputs using Goldschmidt iteration can also be used in the two-party case as all the required sub-protocols can be achieved. Some other protocols, such as BitConj, BitRotate, LTE, Remainder, ShiftL, Eq and PrivShiftL can be built from their prerequisite protocols using commonly known algorithms. The following section explains some of the more advanced computation protocols for two-party unsigned integer arithmetic.

New Protocols for Arithmetic Operations This section describes protocols for less than comparison Alg. 1, division with a public divisor Alg. 2, characteristic vector Alg. 3, exponentiation with a public base Alg. 4 and right shift with private inputs Alg. 5. The protocols considered in this section are compositions of smaller input private protocols and local operations and are therefore themselves input private. For brevity we do not propose individual security theorems for the protocols.
The less than comparison (LT) in Alg. 1 relies on the possibility to extract the most significant bit. Either the most significant bit of the inputs x and y differs and $y[k]$ gives the protocol result or the most significant bits are the same and $(x-y)[k]$ determines the outcome.
Division protocol PubDiv in Alg. 2 is inspired by [16] three-party version. It uses the general idea of finding an inverse of the divisor and multiplying with that rather than dividing. More concretely [29] specifies that there exists a reciprocal c of the divisor as a fixed length number that gives $\lfloor \frac{x}{d} \rfloor = \lfloor \frac{c \cdot x}{2^{2k}} \rfloor$ for $x, d \in \mathbb{Z}_{2^k}$. Alg. 2 works for all values of the divisor d, but can be optimized to use smaller data sizes if d has less than k bits.
A characteristic vector of an element x is a vector $v_0, v_1, \ldots, v_{\ell-1}$ where $v_x = 1$ and the rest of the values are zeros. It can be computed easily using precomputed random element and the corresponding vector as defined in Alg. 3. ChVector protocol is a bit tricky in general

Algorithm 1 LT: $x < y$

Data: Value $[\![x]\!]_{\bmod 2^k}$, $[\![y]\!]_{\bmod 2^k}$
Result: Shared bit $[\![b]\!]_{\bmod 2}$ where $b = 1$ iff $x < y$

1: $[\![x']\!]_{\bmod 2} = [\![x[k]]\!]$ as k'th bit extraction of $[\![x]\!]_{\bmod 2^k}$ using i'th-Bit
2: $[\![y']\!]_{\bmod 2} = [\![y[k]]\!]$ as k'th bit extraction of $[\![y]\!]_{\bmod 2^k}$ using i'th-Bit
3: $[\![d]\!] = [\![x]\!] - [\![y]\!]$
4: $[\![d']\!]_{\bmod 2} = [\![d[k]]\!]$ as k'th bit extraction of $[\![d]\!]_{\bmod 2^k}$ using i'th-Bit
5: $[\![a]\!] = ([\![x']\!] \oplus [\![y']\!]) \wedge [\![y']\!]$
6: $[\![c]\!] = \neg([\![x']\!] \oplus [\![y']\!]) \wedge [\![d']\!]$
7: $[\![b]\!]_{\bmod 2} = [\![a]\!] \oplus [\![c]\!]$

Algorithm 2 PubDiv: Integer division with a public divisor

Data: Value $[\![x]\!]_{\bmod 2^k}$ and public value $d < 2^k$
Result: Shared value $[\![y]\!]_{\bmod 2^k}$ where $y = \lfloor \frac{x}{d} \rfloor$

1: public computation $c = \lfloor \frac{2^{2\cdot k}-1}{d} \rfloor + 1$
2: $[\![x']\!]_{\bmod 2^{3\cdot k}} = Extend([\![x]\!]_{\bmod 2^k})$
3: $[\![u]\!]_i = c \cdot [\![x']\!]_i$
4: $\mathcal{CP}_i$ sets $[\![v]\!]_i = [\![u]\!]_i \gg 2 \cdot k$
5: $\mathcal{CP}_i$ sets $[\![w]\!]_i = [\![u]\!]_i \bmod 2^{2\cdot k}$
6: $[\![y]\!]_{\bmod 2^k} = [\![v]\!]_{\bmod 2^k} + Extend(Overflow([\![w]\!]_{\bmod 2^{2\cdot k}}))$

as it requires taking a modulus which needs an expensive division algorithm to compute the remainder for that modulus. However, in case ℓ is a power of two, this is achieved easily by each party taking the modulus of its share locally. The current use-cases allow to pick ℓ in a way that it is easy to compute this protocol. Note that it is also conceptually easy although inefficient to precompute the required share and vector pair as it can be done by first generating a random element and then computing the characteristic vector using equality checking (Eq) protocol.

Algorithm 3 ChVector: Characteristic vector of a value

Data: Value $[\![x]\!]_{\bmod 2^k}$, public parameter ℓ that limits the actual size of x, precomputed random element $[\![r]\!]_{\bmod 2^k}$ and it's characteristic vector $[\![r_1]\!]_{\bmod 2}, \ldots, [\![r_\ell]\!]_{\bmod 2}$
Result: Shared bitvector $[\![x_1]\!], \ldots, [\![x_\ell]\!]$

1: $[\![z]\!] = [\![x]\!] - [\![r]\!] \bmod \ell$ ▷ Hard if ℓ is not power of 2
2: $z = Declassify([\![z]\!])$
3: return $[[\![r_1]\!]_{\bmod 2}, \ldots, [\![r_\ell]\!]_{\bmod 2}]$ rotation z times to the left

An example protocol that uses ChVector is exponentiation with a public base and secret shared exponent (PubExp). Basically Alg. 4 obliviously chooses the right outcome from all possible results. The same idea can also be applied in the case of private base and exponent, however it would result in a significantly more expensive protocol as Mult protocol would be required instead of public multiplication. The counterpart for exponentiation with a public exponent can be achieved using square-and-multiply.
Public exponentiation is sufficient for computing left shift with a private shifter (PrivShiftL), however, obtaining private right shift from ChVector is more complicated. The core of PrivShiftR is in Alg. 5. The algorithm checks that the shifter is suitably small and in this case applies a sub-protocol for shift when the result might be non-zero. If the shift is larger than the bit

Algorithm 4 PubExp: Exponentiation with a public base

Data: Value $[\![x]\!]_{\bmod 2^k}$ and public value $b \in \mathbb{Z}_{2^\ell}$, parameter $m \leq 2^k$ is the bound on $x < m$
Result: Shared value $[\![y]\!]_{\bmod 2^\ell}$ where $y = b^x \bmod 2^\ell$

1: Public computation of $b, b^2, b^3, \ldots, b^m$ in $\mathbb{Z}_{2^\ell}$
2: $[\![x_1]\!]_{\bmod 2}, \ldots, [\![x_m]\!]_{\bmod 2} = ChVector([\![x]\!], m)$
3: $[\![x_i]\!]_{\bmod 2^\ell} = Extend([\![x_i]\!]_{\bmod 2})$
4: $[\![y]\!] = \sum_{i=1}^{m} b^{i-1} [\![x_i]\!]_{\bmod 2^\ell}$

length of the type, then the result should be zero and this is obtained using multiplication with the indicator variable for the size of the shifter.

Algorithm 5 PrivShiftR*: Shifter size safe private right shift

Data: Value $[\![x]\!]_{\bmod 2^k}$, $[\![p]\!]_{\bmod 2^\ell}$
Result: Shared value $[\![w]\!]_{\bmod 2^k}$ where $w = x \gg p$ is the value of x shifted right by p bits

1: $[\![a]\!]_{\bmod 2^k} = PrivShiftR([\![x]\!]_{\bmod 2^k}, [\![p]\!]_{\bmod 2^\ell})$
2: $[\![b]\!]_{\bmod 2} = LT([\![p]\!]_{\bmod 2^\ell}, Classify(k))$
3: $[\![b]\!]_{\bmod 2^k} = Extend([\![b]\!]_{\bmod 2})$
4: $[\![w]\!]_{\bmod 2^k} = [\![b]\!]_{\bmod 2^k} \cdot [\![a]\!]_{\bmod 2^k}$

The actual shifting algorithm for a suitably small shifter in Alg. 6 is a more complicated protocol. In essence, the idea is to convert to bitwise sharing, perform the shift on those bits and then convert the result back. However, for a private shifter the right shifted outcome has to be chosen obliviously from all possible shifts. In Alg. 6 the shifter value determines which diagonal in the table makes up the real output and the right diagonal is chosen obliviously using the characteristic vector of the shift. For example, for $p = 0$ the output will be exactly the same as the input, but for $p = 2$ the third diagonal is taken so that the least significant bit in the outcome is x_3.

Algorithm 6 PrivShiftR: Private right shift with limited shifter

Data: Value $[\![x]\!]_{\bmod 2^k}$, $[\![p]\!]_{\bmod 2^\ell}$ where $p < k$
Result: Shared value $[\![w]\!]_{\bmod 2^k}$ where $w = x \gg p$ is the value of x shifted right by p bits

1: $[[\![p_1]\!], \ldots, [\![p_k]\!]] = ChVector([\![p]\!], k)$
2: $[[\![x_1]\!], \ldots, [\![x_k]\!]] = BitExtr([\![x]\!])$
3: Compute table T:

$[\![x_1 \wedge p_1]\!]$	$[\![x_2 \wedge p_2]\!]$	$[\![x_3 \wedge p_3]\!]$	$\ldots$	$[\![x_k \wedge p_k]\!]$
0	$[\![x_2 \wedge p_1]\!]$	$[\![x_3 \wedge p_2]\!]$	$\ldots$	$[\![x_k \wedge p_{k-1}]\!]$
0	0	$[\![x_3 \wedge p_1]\!]$	$\ldots$	$[\![x_k \wedge p_{k-2}]\!]$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
0	0	0	$\ldots$	$[\![x_k \wedge p_1]\!]$

with columns T_i

4: $([\![w_1]\!], \ldots, [\![w_k]\!])^T = T_1 \oplus T_2 \oplus \ldots \oplus T_k$
5: Convert $[\![w_1]\!], \ldots, [\![w_k]\!]$ to $[\![w]\!]_{\bmod 2^k}$

Conversion Protocols Aside from the computation protocols, the protocols to convert between different data representations are also required by many of the previously mentioned algorithms. The ideas from ABY framework are still usable to convert between bitwise and general additive secret sharing. However, we also need to consider conversions between various

sizes of additive sharing. In addition, we propose some alternative protocols for conversions already existing in ABY.
Firstly, a special case of the extension protocol is converting a shared bit $[\![x]\!]_{\bmod 2}$ in binary ring $\mathbb{Z}_2$ to a shared bit value $[\![x]\!]_{\bmod 2^k}$ in some larger ring. This bit extension (BitExt) protocol is given in Alg. 7.

Algorithm 7 BitExt: Shared bit to k-bit additive scheme

Data: Shared bits $[\![x]\!]_{\bmod 2}$
Result: Value $[\![x]\!]_{\bmod 2^k}$

1: $\mathcal{CP}_i$ set $[\![u]\!]_{\bmod 2^k}$ as $[\![u]\!]_1 = [\![x]\!]_1$ and $[\![u]\!]_2 = 0$
2: $\mathcal{CP}_i$ set $[\![v]\!]_{\bmod 2^k}$ as $[\![v]\!]_1 = 0$ and $[\![v]\!]_2 = [\![x]\!]_2$
3: $[\![x]\!]_{\bmod 2^k} = [\![u]\!]_{\bmod 2^k} + [\![v]\!]_{\bmod 2^k} - 2 \cdot [\![u]\!]_{\bmod 2^k} \cdot [\![v]\!]_{\bmod 2^k}$

The single bit conversion either using the protocol in Alg. 7 or the OT based conversion from ABY can be used to convert from $[\![x]\!]_{\bmod 2^k}$ to $[\![x]\!]_{\bmod 2^m}$ where the result domain is larger as $m > k$. The idea of the protocol in Alg. 8 is very simple as the same shares can be used with extra care for the case when the sum of the two shares in the initial domain is actually larger than the modulus.

Algorithm 8 Extend: Extension from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^m}$ where $m > k$

Data: $[\![x]\!]_{\bmod 2^k}$
Result: Value $[\![u]\!]_{\bmod 2^m}$ where $u = x$

1: $\mathcal{CP}_i$ sets $[\![u]\!]_i = [\![x]\!]_i$
2: $[\![b]\!]_{\bmod 2} = Overflow([\![x]\!])$
3: $[\![b']\!]_{\bmod 2^m} = BitExt([\![b]\!]_{\bmod 2})$
4: $[\![u]\!] = [\![u]\!] - 2^k \cdot [\![b']\!]$

The opposite operation to extension is truncation (Trunc) to a smaller domain from $[\![x]\!]_{\bmod 2^m}$ to $[\![x \bmod 2^k]\!]_{\bmod 2^k}$ where $m > k$. Converting $\mathbb{Z}_{2^m}$ to $\mathbb{Z}_{2^k}$ can be obtained easily by both parties computing $[\![x]\!]_i \bmod 2^k$ and setting the result as their output share. Differently from the extension this method also applies for a single bit outcome. In this case the protocol also has a very clear meaning of finding the least significant bit of the shared element.
Although the least significant bit can be easily found using simple local truncation, finding any other single bit is more complicated. Alg. 9 defines general bit extraction from the additively shared element by taking care of possible overflows from the less significant bits of the shares.

Algorithm 9 i'th-Bit: Extract ℓ'th bit of shared value

Data: Value $[\![x]\!]_{\bmod 2^k}$, public positive integer $1 \leq \ell < k$
Result: Shared ℓ'th bit $[\![b]\!]_{\bmod 2}$ of x as $b = x[\ell]$

1: **if** $\ell = 1$ **then**
2: $\quad \mathcal{CP}_i$ sets $[\![b]\!]_i = [\![x]\!]_i[1]$ as the least significant bit of the share
3: **else**
4: $\quad \mathcal{CP}_i$ sets $[\![x']\!]_i = [\![x]\!]_i \bmod 2^{\ell-1}$ ▷ $[\![x']\!]_{\bmod 2^{\ell-1}}$
5: $\quad [\![b']\!]_{\bmod 2} = Overflow([\![x']\!]_{\bmod 2^{\ell-1}})$
6: $\quad \mathcal{CP}_i$ sets $[\![m]\!]_i = [\![x]\!]_i[\ell]$ as the ℓ'th bit of their share
7: $\quad [\![b]\!] = [\![m]\!] \oplus [\![b']\!]$
8: **end if**

Besides single bit extraction it is also possible to obtain a full bit representation of the shared integer value. Bit extraction protocols have been discussed by ABY, but for completeness we include a version that is based solely on additive secret sharing. We denote $x = \sum_{i=1}^{k} 2^{i-1} x_i$ where $x_1, \ldots, x_k$ is the bit decomposition of the value x. The idea behind bit-decomposition (BitExtr) is that both parties classify each bit of their share $[\![x]\!]_i$ and then evaluate a binary addition circuit to obtain the bit decomposition of the shared element that is the sum of the two shares. The addition circuit performs carry-lookahead addition in Alg. 11 similarly to [14] where BitExtr discards the highest bit.

Algorithm 10 Carry: Carry bits in bitwise addition

Data: Shared bit-vectors $[\![x_1]\!]_{\bmod 2}, \ldots [\![x_b]\!]_{\bmod 2}$ and $[\![y_1]\!]_{\bmod 2}, \ldots [\![y_b]\!]_{\bmod 2}$
Result: Shared bit-vector $[\![s_1]\!]_{\bmod 2}, \ldots [\![s_b]\!]_{\bmod 2}$, where s are the carry bits that occur when adding x and y

$[\![p_i]\!] = [\![x_i]\!] \oplus [\![y_i]\!]$ ▷ Flags for carry propagation
$[\![s_i]\!] = [\![x_i]\!] \cdot [\![y_i]\!]$
for $j \in \{0, \ldots, \log_2 b - 1\}$ **do**
  for $\ell \in \{0, \ldots, 2^j - 1\}$ **do**
    for $m \in \{0, \ldots, \frac{b}{2^{j+1}} - 1\}$ **do**
      $t = 2^j + \ell + 2^{j+1} m + 1$
      $d = 2^j + 2^{j+1} m$
      $[\![s_t]\!] = [\![s_t]\!] \oplus ([\![p_t]\!] \cdot [\![s_d]\!])$
      $[\![p_t]\!] = [\![p_t]\!] \cdot [\![p_d]\!]$
    end for
  end for
end for

Algorithm 11 Binary addition circuit

Data: Shared bit-vectors $[[\![x_1]\!]_{\bmod 2}, \ldots [\![x_b]\!]_{\bmod 2}]$, $[[\![y_1]\!]_{\bmod 2}, \ldots [\![y_b]\!]_{\bmod 2}]$
Result: Shared bit-vector $[[\![w_1]\!]_{\bmod 2}, \ldots [\![w_{b+1}]\!]_{\bmod 2}]$, where $w = x + y$

$[[\![s_1]\!], \ldots, [\![s_b]\!]] \leftarrow Carry([[\![x_1]\!], \ldots [\![x_b]\!]], [[\![y_1]\!], \ldots [\![y_b]\!]])$
$[\![w_1]\!] = [\![y_1]\!] \oplus [\![x_1]\!]$
for $i \in \{2, \ldots, b\}$ **do**
  $[\![w_i]\!] = [\![y_i]\!] \oplus [\![x_i]\!] \oplus [\![s_{i-1}]\!]$
end for
$[\![w_{b+1}]\!] = [\![s_b]\!]$

3.5 Zero-Knowledge from Garbled Circuits (and GC for ZK)

Zero-knowledge protocols are one of the fundamental concepts in modern cryptography and have countless applications. However, after more than 30 years from their introduction, there are only very few languages (essentially those with a group structure) for which we can construct zero-knowledge protocols that are efficient enough to be used in practice. This is problematic, since zero-knowledge protocols (ZK) are one of the main building blocks in constructing MPC protocols which are secure against malicious corruptions.

The paper described here [26] describes work that was done in AU and proposes a solution to this problem. It is based on a slightly earlier work of AU and SAP [35] that presented a protocol based on Yao's garbled circuit technique that supports efficient zero-knowledge proofs for generic languages (e.g., to prove statements of the form "I know x s.t. $y = \text{SHA-256}(x)$" for a common input y). The new work in [26] shows that garbled circuits can be optimized for this specific application.

3.5.1 Zero-Knowledge Vs. Generic 2PC

It is clear that zero-knowledge is a subset of secure two-party computation (i.e., any protocol for generic secure computation can be used to do zero-knowledge, including those based of garbled circuits). The main contribution of [35] is to construct an efficient protocol for the special case of secure two-party computation where only one party has input (like in the zero-knowledge case). The protocol achieves active security and is essentially only twice as slow as the passive secure version of Yao's garbled circuit protocol. This is a great improvement with respect to the *cut-and-choose* technique to make Yao's protocol actively secure, where the complexity grows linearly with the security parameter.

3.5.2 Zero-Knowledge From Garbled Circuits

In Figure 3.4 a sketch of the ZK protocol proposed by Jawurek *et al.* [35] is shown. The protocol proceeds as follows: The prover (acting as the receiver in the OT) uses the bits of his witness x as choice bits in the OT while the verifier (acting as the sender in the OT) uses as input all the pairs of keys of the garbled circuits. The verifier also sends the garbled circuit F. Now if the prover uses a valid witness, he can evaluate the garbled circuit and compute the output key corresponding to the output bit 1. However, instead of disclosing this key at this stage, the prover commits to it and waits for the verifier to prove that she constructed the garbled circuit correctly (and acted honestly in the OT protocols as well). If this check goes through, the prover opens the commitment and the verifier accepts the proof if the commitment contains the key corresponding to the output bit 1. The main ideas behind the proof of security in [35] are as follows: soundness (the verifier accepts only if the statement is true) is achieved thanks to the *authenticity* property of garbled circuits – using the terminology of Bellare *et al.* [11]. At the same time the protocol is zero-knowledge (the verifier learns *only* that the statement is true) because the prover verifies that she generates the GC honestly before disclosing any information.

3.5.3 Garbled Circuits for Zero-Knowledge

In the last few years garbled circuits have been elevated from being merely a component in Yao's protocol for secure two-party computation, to a cryptographic primitive in its own right, following the growing number of applications that use GCs, including the zero-knowledge example described above.
In [26] we have shown that due to the property of this particular application (i.e., one of the parties knows all the secret input bits, and therefore all intermediate values in the computation), we can construct more efficient garbling schemes specifically tailored to this goal.
As a highlight of the results in [26], in one of the constructions only *one ciphertext* per gate needs to be communicated and XOR gates never require any cryptographic operations. In addition to making a step forward towards more practical ZK, we believe that this contribution is also interesting from a conceptual point of view: in the terminology of Bellare *et al.* [11] these

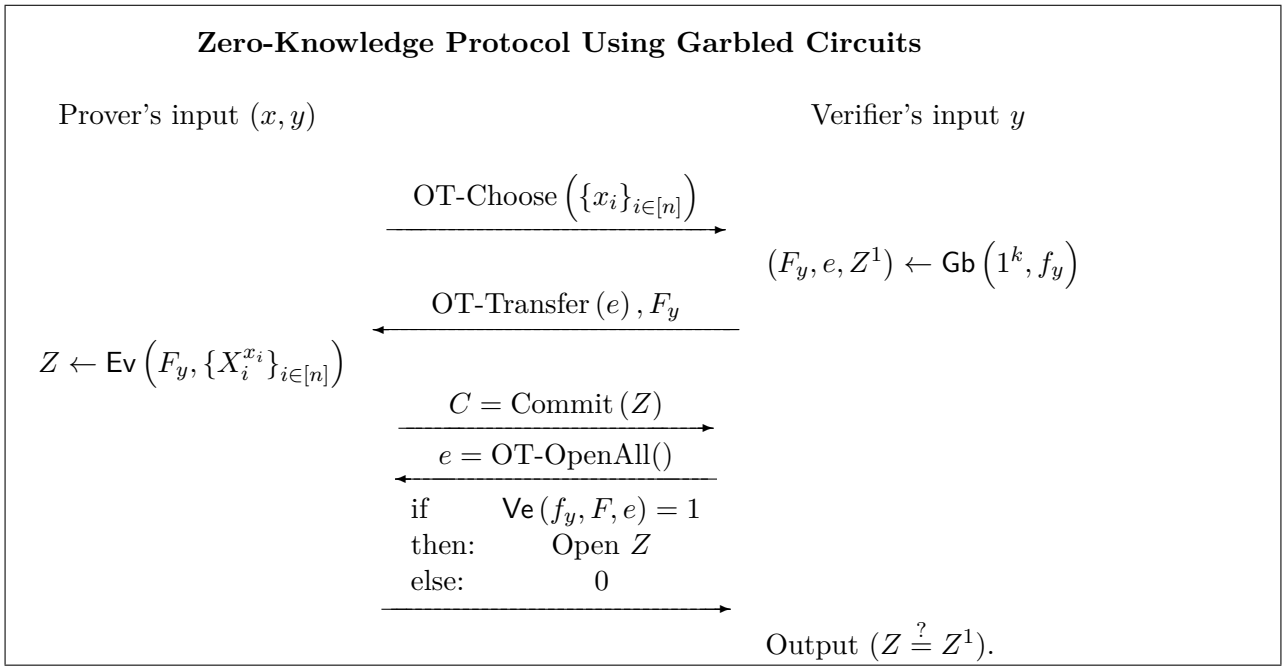


Figure 3.4: Informal Description of Jawurek *et al.* ZK from GC.

garbling schemes achieve authenticity, but no privacy nor obliviousness, therefore representing the first *natural* separation between those notions.
From a technical point of view, one of the main properties of Yao's garbling scheme is that the circuit evaluator cannot learn the values associated to the internal wires during the evaluation of the garbled circuit. This implies that the evaluation of each garbled gate must be *oblivious* (it must be the same for each input combination). The garbling schemes presented here give up on this property and allow the evaluator to learn the values associated which each wire in the circuit, who can explicitly use this knowledge to perform *non-oblivious* garbled gate evaluation. This allows to reduce significantly the size of a garbled circuit and the computational overhead for the circuit constructor. This does not have any impact on *authenticity* i.e., the only thing that a malicious evaluator can do with a garbled input and a garbled circuit is to use them in the intended way, that is to evaluate the garbled circuit on the garbled input and produce the (correct) garbled output.
The new garbling schemes can be immediately plugged-in in the Zero-Knowlkedge Protocol from the previous section, thus making it even more practical.

3.5.4 Overview of The Garbling Schemes

In a nutshell, the proposed garbling schemes work as follows: Consider a NAND gate, with associate input keys L^0, L^1, R^0, R^1 for the left and right wire respectively, and output keys O^0, O^1. The circuit constructor needs to provide the evaluator with a cryptographic gadget that, on input L^a, R^b, outputs the corresponding output key $O^{a\bar{\wedge}b}$. Remember that our goal is not privacy, but only authenticity, meaning that the evaluator is allowed to learn a and b but even a corrupted evaluator should not learn $O^{1-(a\bar{\wedge}b)}$. In particular, this means that the evaluator should learn O^0 if and only if (iff) he holds both L^1 and R^1. This can be ensured by encrypting O^0 under *both* L^1 and R^1.
On the other hand, it is enough that one of the inputs is 0 for the output to be 1, so it "should be enough" to hold L^0 or R^0 to learn O^1. In standard Yao GCs we do not want the evaluator to

learn which of the three possible combinations of input keys he owns between (L^0, R^0), (L^0, R^1) and (L^1, R^0) (nor the output of the gate) and therefore we encrypt O^1 under all the three possibilities in the same way as we encrypt the 0 key. But if the evaluator is allowed to know which bits keys correspond to, we can simply encrypt O^1 separately under L^0 and R^0, thus saving one encryption.
Note that, using the row-reduction technique, we can instead derive O^0 as $O^0 = \mathsf{KDF}(L^1, R^1)$ and therefore we can remove one ciphertext from the garbled table. We now have two-choices:

- If we want to be compatible with the free-XOR technique the value O^1 is already determined by O^0 and the global difference Δ, and thus no more row-reduction is possible.
- Alternatively we can decide to give up on free-XOR and derive O^1 as $O^1 = \mathsf{KDF}(L^0)$, thus removing yet another ciphertext from the garbled table, that now contains only the ciphertext $C = O^1 \oplus \mathsf{KDF}(R^0)$. In this case, we garble XOR gates as follows: We define the output keys O^0 and O^1 respectively as $O^0 = L^0 \oplus R^0$ and $O^1 = L^0 \oplus R^1$ and we reveal the value $C = L^0 \oplus R^0 \oplus L^1 \oplus R^1$. Due to the symmetry of the XOR gate, now the evaluator can always derive the correct output key. Note that now XOR gates do not require any cryptographic operation but only the communication of a k-bit string (k being the security parameter).

Chapter 4

Order-Preserving Encryption for Secure Database Qeuries

4.1 Optimal Average-Complexity Ideal-Security Order-Preserving Encryption

4.1.1 Introduction

Order-preserving encryption enables performing many classes of queries – including range queries – on encrypted databases. Popa et al. recently presented an ideal-secure order-preserving encryption (or encoding) scheme [55], but their cost of insertions (encryption) is very high. Kerschbaum et al. presented an also ideal-secure, but significantly more efficient order-preserving encryption scheme [40]. This scheme is inspired by Reed's referenced work on the average height of random binary search trees. And the scheme improves the average communication complexity from $O(n \log n)$ to $O(n)$ under uniform distribution. Kerschbaum et al.'s scheme also integrates efficiently with adjustable encryption as used in CryptDB. In their experiments for database inserts Kerschbaum et al. achieve a performance increase of up to 81% in LANs and 95% in WANs.

4.1.2 Scheme

Kerschbaum et al.'s order-preserving encryption algorithm builds a binary search tree as does Popa et al.'s. Kerschbaum et al's is however not necessarily balanced and relies on the uniformity assumption about the input distribution. They only balance the tree when necessary, i.e., then an update operation is performed. This enables them to maintain the dictionary on the client and therefore achieve a significant performance gain and compatibility with adjustable onion encryption.

Consider the following example: $N = 16$ and $M = 256$. Let $n = 5$, $x_1 = 13$, $x_2 = 5$, $x_3 = 7$, $x_4 = 5$ and $x_5 = 12$. Note that $x_2 = x_4$ is a duplicate. Then $m = 4$, $y_1 = 128$, $y_2 = 64$, $y_3 = 96$ and $y_5 = 112$ (without necessity for any ciphertext modification). For this ordered sequence we have $j_1 = 2$, $j_2 = 3$, $j_3 = 5$, and $j_4 = 1$, i.e., $x_{j_1} = 5$, $y_{j_1} = 64$ and so on. The corresponding binary search tree is visualized in Figure 4.1.

The input to the encryption algorithm is a plaintext x_i. Encryption is stateful and stores an ordered list of plaintext-ciphertext pairs $\langle x_i, y_i \rangle$. This list is initialized to $\langle -1, -1 \rangle$, $\langle N, M \rangle$. The output of the encryption, i.e. the ciphertext y_i, is sent to the database server.

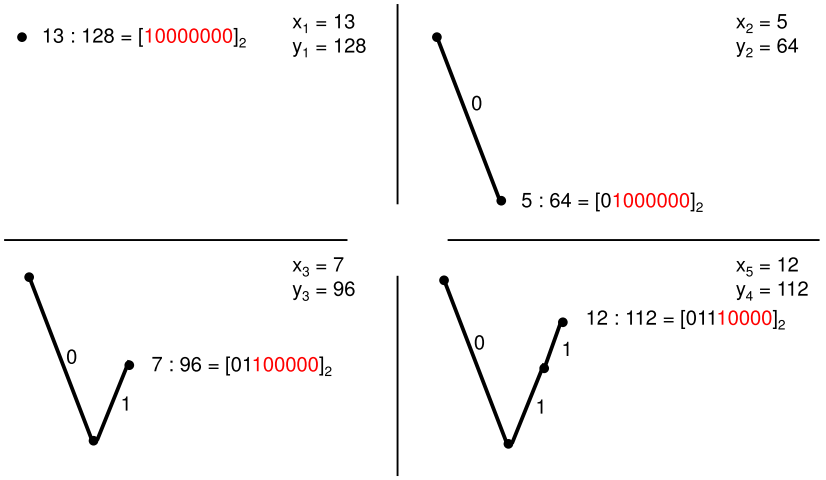


Figure 4.1: Search Trees for Insertion of 13, 5, 7, 12

We emphasize that the encryption algorithm is keyless. The state of the algorithm plays the role of the key, i.e. it is secret information. The size of the state of the encryption algorithm is the size of the dictionary of the database. It is therefore not necessary to keep a copy of the data, but only of the dictionary. One can hence reap the same size benefits as dictionary compression (roughly over a factor of 20 [54] which is already achieved for the dictionary and the data identifier column).
The update algorithm potentially updates all ciphertexts produced so far. It re-encrypts all (distinct) plaintexts in order, i.e. the median element first and so on. Thus, it produces a (temporarily) balanced tree.
The state of the encryption algorithm is updated on the database client. This updated state needs to be sent to the database server and its persistent data needs to be updated – potentially all database rows. This affects not only the column store, but also the entire dictionary.
Finally, the decryption algorithm is a simple lookup in the state.

4.2 Frequency-Hiding Order-Preserving Encryption

4.2.1 Introduction

Kerschbaum et al. present a scheme that achieves a strictly stronger notion of security than any other order-preserving encryption scheme so far [39]. The basic idea is to randomize the ciphertexts to hide the frequency of plaintexts. Still, the client storage size remains small, in their experiments up to 1/15 of the plaintext size. As a result, one can more securely outsource large data sets, since Kerschbaum et al. also show that their security increases with larger data sets. They clearly increase security while preserving the functionality for most queries relying on the ordering information. However, they also increase client storage size and introduce a small error in some queries. Kerschbaum et al. present a definition of a new, stronger security notion for order-preserving encryption than indistinguishability under chosen plaintext attack which they call *indistinguishability under frequency-analyzing ordered chosen plaintext attack*. They also present a scheme implementing this notion including compression mechanisms.

4.2.2 Scheme

Kerschbaum et al. initially proceed as the deterministic order-preserving encryption scheme of [40] and insert plaintexts into a sorted binary tree in the order they are encrypted. They then handle plaintexts that have already been encrypted differently. In this case, i.e. when inserted plaintext and to be encrypted plaintext are equal, they traverse the tree in a randomly chosen fashion and insert the new plaintext as a leaf.

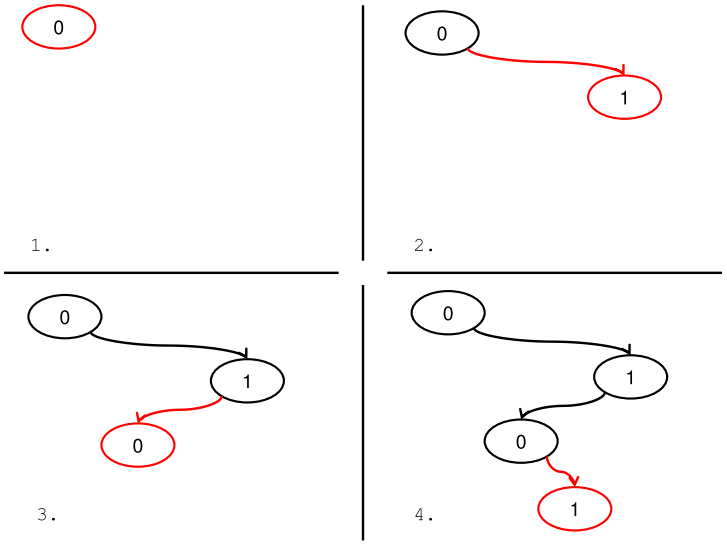


Figure 4.2: Growing Search Tree for Sequence $0, 1, 0, 1$

We consider the example of a binary plaintext domain, e.g. male and female. We have 4 plaintexts $x_i \in \{0, 1\}$. We insert the following sequence $X = 0, 1, 0, 1$. We set the random coins to the sequence $1, 0$. The resulting sequence of trees is depicted in Figure 4.2.
Figure 4.2 is divided into four subfigures numbered 1 to 4. Each depicts the search tree after inserting one more element of the sequence with the new node in red. In subfigure 3 we see for the first time a plaintext repeating, but inserted beneath a parent with a different plaintext. We can trace the algorithm as follows: When inserting 0 for the second time, the algorithm encounters a 0 at the root. Due to the random coins it traverses to the right, where it encounters a 1 and must make a deterministic choice leading to the new leaf. In subfigure 4 we see that the next 1 inserted and plaintext nodes interleaving. In larger plaintext domains even intermediate elements can be placed at lower nodes.
Of course, repeated plaintexts can also be placed under parents with the same plaintext. If we insert two more elements $0, 1$ with random coins $0, 1$, the search tree will look as in Figure 4.3.

4.2.3 Compression

Finally, the tree is compressed. We can compress the plaintext using regular dictionary compression [1, 13, 63] and store repeated values as the index into the dictionary. Moreover, we can further compress subtrees of repeated values. Kerschbaum et al. call subtrees of the same plaintext *clusters*. In a cluster we do not need to store the plaintext for each node, instead we just store it once in the root of the cluster. Instead of storing the tree structure we only store its traversal thereby compressing the size of the pointers.

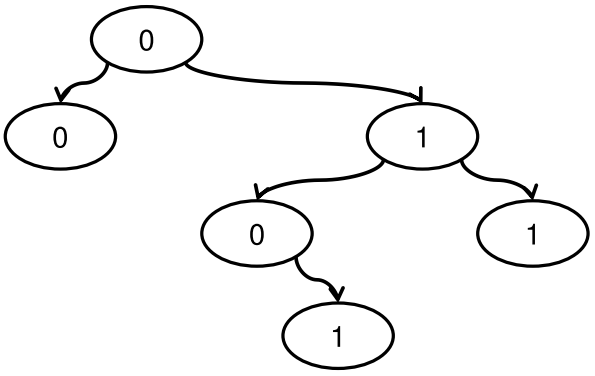


Figure 4.3: Possible Search Tree for Sequence $0, 1, 0, 1, 0, 1$

Chapter 5

Protocols for Private Set Intersection

Private set intersection (PSI) allows two parties $\mathcal{A}$ and $\mathcal{B}$ with respective input sets X and Y to compute the intersection $X \cap Y$ of their sets without revealing any information but the intersection itself. Although PSI has been widely studied in the literature, many real-world applications today use an insecure hash-based protocol instead of a secure PSI protocol, mainly because of the insufficient efficiency of current PSI protocols.
In a sequence of works we presented improved PSI protocols that were more efficient by the state of the art by at least an order of magnitude. The most advanced family of protocols that we presented, denoted *Phasing* for Permutation-based Hashing Set Intersection, is a new approach for constructing PSI protocols based on a hashing technique that ensures that hashed elements can be represented by short strings without any collisions. The overhead of recent PSI protocols depends on the length of these representations, and this new structure of construction, together with other improvements, results in very efficient performance that is only moderately larger than that of the *insecure* protocol that is in current real-world usage.

5.1 Contributions

The goal of this work was to enable PSI computations for large scale sets that were previously beyond the capabilities of state-of-the-art protocols. The constructions that were designed improve performance by more than an order of magnitude. These improvements were obtained by generalizing the hashing approach of [51] and applying it to generic secure computation-based PSI protocols. The hash function in [51] were replaced by a permutation which enables to reduce the bit-length of internal representations. Moreover, several improvements to the OT-based PSI protocol of [51] were presented. The contributions are next explained in more detail:

Phasing: Using permutation-based hashing to reduce the bit-length of representations. The overhead of the best current PSI protocol [51] is linear in the length of the representations of items in the sets (i.e., the ids of items in the sets). The protocol maps items into bins, and since each bin has very few items in it, it is tempting to hash the ids to shorter values and trust the birthday paradox to ensure that no two items in the same bin are hashed to the same representation. However, a closer examination shows that to ensure that the collision probability is smaller than $2^{-\lambda}$, the length of the representation must be at least λ bits, which is too long.
The new results utilize the permutation-based hashing techniques of [2] to reduce the bit-length of the ids of items that are mapped to bins. These ideas were suggested in an algorithmic setting

to reduce memory usage, and our new work is the first time that they are used in a cryptographic or security setting to improve performance. Essentially, when using β bins the first $\log \beta$ bits in an item's hashed representation define the bin to which the item is mapped, and the other bits are used in a way which provably prevents collisions. This approach reduces the bit-length of the values used in the PSI protocol by $\log \beta$ bits, and this yields reduced overhead by up to 60%-75% for the settings we examined.

Circuit-Phasing: Improved circuit-based PSI. There is a great advantage in using generic secure computation for computing PSI, since this enables to easily compute variants of the basic PSI functionality. Generic secure computation protocols evaluate Boolean circuits computing the desired functionality. The best known circuit for computing PSI was based on the Sort-Compare-Shuffle circuit of [31]. The new work describes Circuit-Phasing, a new generic protocol that uses hashing (specifically, Cuckoo hashing and simple hashing) and secure circuit evaluation. In comparison with the previous approach, the new circuits have a smaller number of AND gates, a lower depth of the circuit (which affects the number of communication rounds in some protocols), and a much smaller memory footprint. These factors lead to a significantly better performance.

OT-Phasing: Improved OT-based PSI. The new work introduces the OT-Phasing protocol which improves the OT-based PSI protocol of [51] as follows:

- **Improved computation and memory.** The length of the strings that are processed in the OT is reduced from $\mathsf{O}(\log^2 n)$ to $\mathsf{O}(\log n)$, which results in a reduction of computation and memory complexity for the client from $\mathsf{O}(n \log^2 n)$ to $\mathsf{O}(n \log n)$.
- **3-way Cuckoo hashing.** The construction uses 3 instead of 2 hash functions to generate a more densely populated Cuckoo table and thus decrease the overall number of bins and hence OTs.
- **Faster OT extension.** The construction implements OT extension using fixed-key AES hashing instead of the SHA hash function. This change improves the overhead since it enables to use the AES-NI instruction.

OT-Phasing improves over state-of-the-art PSI both in terms of run-time and communication. Compared to the previously fastest PSI protocol of [51], the new protocol improves run-time by up to factor 10 in the WAN setting and by up to factor 20 in the LAN setting. Furthermore, the new OT-Phasing protocol in some cases achieves similar communication as [49], which was shown to achieve the lowest communication of all PSI protocols [51].

5.2 Evaluation

We report on our empirical performance evaluation of Circuit-Phasing and OT-Phasing schemes suggested in [52]. We evaluate their performance separately (§5.2.1 and §5.2.2), since special purpose protocols for set intersection were shown to greatly outperform circuit-based solutions in [51]. (The latter are nevertheless of independent interest because their functionality can be easily modified.)

Benchmarking Environment We consider two benchmark settings: a *LAN* setting and a *WAN* setting. The LAN setting consists of two desktop PCs (Intel Haswell i7-4770K with 3.5 GHz and 16GB RAM) connected by Gigabit LAN. The WAN setting consists of two Amazon EC2 m3.medium instances (Intel Xeon E5-2670 CPU with 2.6 GHz and 3.75 GB RAM) located in the US east coast (North Virginia) and Europe (Frankfurt) with an average bandwidth of 50 MB/s and average latency (round-trip time) of 96 ms.
We perform all experiments for a symmetric security parameter $\kappa = 128$-bit and statistical security parameter $\sigma = 40$, using a single thread (except for GMW, where we use two threads to compute OT extension), and average the results over 10 executions. In our experiments, we frequently encountered outliers in the WAN setting with more than twice of the average run-time, for which we repeated the execution. The resulting variance decreased with increasing input set size; it was between $0.5\% - 8.0\%$ in the LAN setting and between $4\% - 16\%$ in the WAN setting. Note that all machines that we perform our experiments on are equipped with the AES-NI extensions which allows for very fast AES evaluation.

Implementation Details We instantiate the random oracle, the function for hashing into smaller domains, and the correlation-robust function in OT extension with SHA256. We instantiate the pseudo-random generator using AES-CTR and the pseudo-random permutation in the server-aided protocol of [36] using AES. To compute the $2^\mu \times OT_1 t\ell$ functionality, we use the random 1-out-of-N OT extension of [42] and set $\mu = 8$, i.e., use $N = 256$, since this was shown to result in minimal overhead in [51]. We measure the times for the function evaluation including the cost for precomputing the OT extension protocol and build on the OT extension implementation of [3]. Our OT-Phasing implementation is available online at https://github.com/encryptogroup/PSI and our Circuit-Phasing implementation is available as part of the ABY framework of [24] at https://github.com/encryptogroup/ABY.
For Cuckoo hashing, we set $\epsilon = 0.2$ and map n elements to $2(1+\epsilon)n$ bins for 2-way Cuckoo hashing and to $(1+\epsilon)n$ bins for 3-way Cuckoo hashing with a stash size that was set to keep the failure probability low.
For OT-based PSI [51] and OT-Phasing, where the performance depends on the bit-length of elements, we hash the σ-bit input elements into a $\ell = \sigma + \log_2(n_1) + \log_2(n_2)$-bit representation using SHA256 if $\sigma > \ell$. This decrease the performance impact of the bit-length.
We use a Yao's garbled circuits implementation with most recent optimizations, including the recent half-gate optimization of [62].
We emphasize that all implementations are done in the same programming language (C++), use the same underlying libraries for evaluating cryptographic operations (OpenSSL for symmetric cryptography and Miracl for elliptic curve cryptography), perform the plaintext-intersection of elements using a standard hash map, are all executed using a single thread (except for the GMW implementation which uses two threads), and run in the same benchmarking environment.

Protocol	LAN				WAN			
	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
Yao's garbled circuits [60]								
SCS [31]	309	3,464	63,857	—	2,878	20,184	301,512	—
Circuit-Phasing	376	3,154	39,785	—	3,004	17,133	178,865	—
Goldreich-Micali-Wigderson [28]								
SCS [31]	626	2,175	38,727	—	11,870	21,030	218,378	—
Circuit-Phasing	280	1,290	14,149	168,397	2,681	8,681	81,534	846,510

Table 5.1: Run-time in ms for generic secure PSI protocols in the LAN and WAN setting on $\sigma = 32$-bit elements.

Protocol	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	Asymptotic
Number of AND gates					
SCS [31]	229,120	5,238,784	107,479,009	*2,000,000,000	$\sigma(3n\log_2(n) + 4n)$
Circuit-Phasing	297,852	3,946,776	49,964,540	600,833,968	$(\sigma - \log_2(n) - 2)(6(1+\epsilon)n\frac{\ln n}{\ln\ln n} + sn)$
Communication in MB for Yao's garbled circuits [60] and GMW [28]					
SCS [31]	7	169	3,485	*64,850	$2\kappa\sigma(3n\log_2(n) + 4n)$
Circuit-Phasing	9	122	1,550	18,736	$2\kappa(\sigma - \log_2(n) - 2)(6(1+\epsilon)n\frac{\ln n}{\ln\ln n} + sn)$
Number of communication rounds for GMW [28]					
SCS [31]	85	121	157	193	$(\log_2(\sigma) + 4)\log_2(2n) + 4$
Circuit-Phasing	5	5	5	5	$\log_2(\sigma)$

Table 5.2: Number of AND gates, concrete communication in MB, round complexity, and failure probability for generic secure PSI protocols on $\sigma = 32$-bit elements. Numbers with * are estimated.

5.2.1 Generic Secure Computation-based PSI Protocols

For the generic secure computation-based PSI protocols, we perform the evaluation on a number of elements varying from 2^8 to 2^{20} and a fixed bit-length of $\sigma = 32$-bit. For $n = 2^{20}$ all implementations, except Circuit-Phasing with GMW, exceeded the available memory, which is due to the large number of AND gates in the SCS circuit (estimated 2 billion AND gates) and the requirement to represent bits as keys for Circuit-Phasing with Yao, where storing only the input wire labels to the circuit requires 1 GB. A more careful implementation, however, could allow the evaluation of these circuits. We compare the sort-compare-shuffle (SCS) circuit of [31] and its depth-optimized version of [51], with Circuit-Phasing, by evaluating both constructions using Yao's garbled circuits protocol [60] and the GMW protocol [28] in the LAN and WAN setting. We use the size-optimized version of the SCS circuit in Yao's garbled circuit and the depth-optimized version of the circuit in the GMW protocol. The run-time of Circuit-Phasing would increase linear in the bin size max_β, while the stash size s would have a smaller impact on the total run-time as the concrete factors are smaller.

Run-Time (Table 5.1) Our main observation is that Circuit-Phasing outperforms the SCS circuit of [31] for all parameters except Yao's garbled circuits with small set sizes $n = 2^8$. In this case, the high stash size of $s = 12$ greatly impacts the run-time of Circuit-Phasing. When evaluated using Yao's garbled circuits, Circuit-Phasing outperforms the SCS circuit by a factor of 1-2, and when evaluated using GMW it outperforms SCS by a factor of 2-5. Furthermore, the run-time for Circuit-Phasing grows slower with n than for the SCS circuit for all settings except for GMW in the WAN setting. There, the run-time of the SCS circuit grows slower than that of Circuit-Phasing. This can be explained by the high number of communication rounds of the SCS based protocol, which are slowly being amortized with increasing values of n. The slower increase of the run-time of Circuit-Phasing with increasing n is due to the smaller increase of the bin size $max_\beta \in \mathsf{O}(\frac{\ln n}{\ln\ln n})$ vs. $\mathsf{O}(\log n)$ for the SCS circuit, and the use of permutation-based hashing, which reduces the bit-length of the inputs to the circuit. Note that our Yao's garbled circuits implementation suffers from similar performance drawbacks in the WAN setting as our GMW implementation, although being a constant round protocol. This can be explained by the pipelining optimization we implement, where the parties pipeline the garbled circuits generation and evaluation. The performance drawback could be reduced by using an implementation that uses independent threads for sending / receiving.

Communication (Table 5.2) Analogously to the run-time results, Circuit-Phasing improves the communication of the SCS circuit by factor of 1-4 and grows slower with increasing values

of n. The improvement of the round complexity, which is mostly important for GMW, is even more drastic. Here, Circuit-Phasing outperforms the SCS circuit by a factor of 16-38. Note that the round complexity of Circuit-Phasing only depends on the bit-length of items and is independent of the number of elements.

Setting	**LAN**					**WAN**			
Protocol	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	$n = 2^{24}$	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
Naive Hashing$^{(*)}$	1	4	48	712	13,665	97	111	558	3,538
Server-Aided$^{(*)}$ [36]	1	5	78	1,250	20,053	198	548	2,024	7,737
DH-based ECC [49]	231	3,238	51,380	818,318	13,065,904	628	10,158	161,850	2,584,212
Bit-length $\sigma = 32$-bit									
OT PSI [51]	184	216	3,681	62,048	929,685	957	1,820	9,556	157,332
OT-Phasing	179	202	437	4,260	46,631	912	1,590	3,065	14,567
Bit-length $\sigma = 64$-bit									
OT PSI [51]	201	485	7,302	125,697	—	977	1,873	18,998	315,115
OT-Phasing	180	240	865	10,128	137,036	1,010	1,780	5,009	29,387
Bit-length $\sigma = 128$-bit									
OT PSI [51]	201	485	8,478	155,051	—	980	1,879	21,273	392,265
OT-Phasing	181	240	915	13,485	204,593	1,010	1,780	5,536	37,422

Table 5.3: Run-time in ms for protocols with $n = n_1 = n_2$ elements. (Protocols with $^{(*)}$ are in a different security model.)

Protocol	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	$n = 2^{24}$	**Asymptotic [bit]**
Naive Hashing$^{(*)}$	0.01	0.03	0.56	10.0	176.0	$n_1\ell$
Server-Aided$^{(*)}$ [36]	0.01	0.16	2.5	40.0	640.0	$(n_1 + n_2 + \lvert X \cap Y \rvert)\kappa$
DH-based ECC [49]	0.02	0.28	4.56	74.0	1,200.0	$(n_1 + n_2)\varphi + n_1\ell$
Bit-length $\sigma = 32$-bit						
OT PSI [51]	0.09	1.39	22.58	367.20	5,971.20	$0.6n_2\sigma\kappa + 6n_1\ell$
OT-Phasing	0.06	0.73	8.74	136.8	1,494.4	$2.4n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(1.2n_2)\rfloor}{8}\rceil) + (3+s)n_1\ell$
Bit-length $\sigma = 64$-bit						
OT PSI [51]	0.14	2.59	41.78	674.4	10,886.4	$0.6n_2\kappa * min(\ell, \sigma) + 6n_1\ell$
OT-Phasing	0.09	1.34	18.34	290.4	3,952.0	$2.4n_2\kappa(\lceil\frac{min(\ell,\sigma)-\log_2(n_2)}{8}\rceil) + (3+s)n_1\ell$
Bit-length $\sigma = 128$-bit						
OT PSI [51]	0.14	2.59	46.58	828.0	14,572.8	$0.6n_2\ell\kappa + 6n_1\ell$
OT-Phasing	0.09	1.34	20.74	367.2	5,795.2	$2.4n_2\kappa(\lceil\frac{\ell-\log_2(n_2)}{8}\rceil) + (3+s)n_1\ell$

Table 5.4: Communication in MB for PSI protocols with $n = n_1 = n_2$ elements. $\ell = \sigma + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n$ for TTP-based protocol. (Protocols with $^{(*)}$ are in a different security model.)

5.2.2 Special Purpose PSI Protocols

For the special purpose PSI protocols we perform the experimental evaluation for equally sized sets $n_1 = n_2$ and differently sized sets $n_2 \ll n_1$, for set sizes ranging from 2^8 to 2^{24} in the LAN setting and from 2^8 to 2^{20} in the WAN setting.

We compare OT-Phasing to the original OT-based PSI protocol of [51], the naive insecure hashing solution, the semi-honest server-aided protocol of [36], and the Diffie-Hellmann (DH)-based protocol of [49] using elliptic curves. Note that the naive hashing protocol and the server-aided protocol of [36] have different security assumptions and cannot directly be compared to the remaining protocols. We nevertheless included them in our comparison to serve as a baseline on the efficiency of PSI. For the protocol of [36], we run the server routine that computes the intersection between the sets on the machine located at the US east coast (North Virginia) and the server and client routine on the machine in Europe (Frankfurt). For the original OT-based PSI and OT-Phasing, we give the run-time and communication for three bit-lengths:

short $\sigma = 32$ (e.g., for IPv4 addresses), *medium* $\sigma = 64$ (e.g., for credit card numbers), and *long* $\sigma = 128$ (for set intersection between arbitrary inputs).
Note that the OT-based PSI protocol of [51] and our OT-Phasing protocol both evaluate public-key cryptography during the base-OTs, which dominates the run-time for small sets. However, these base-OTs only need to be computed once and can be re-used over multiple sessions. In the LAN setting, the average run-time for computing the 256 base-OTs was 125 ms while in the WAN setting the run-time was 245 ms. Nevertheless, our results all contain the time for the base-OTs to provide an estimation of the total run-time.

Experiments with Equal Input Sizes

In the experiments for input sets of equal size $n = n_1 = n_2$ we set $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ in the LAN setting and $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$ in the WAN setting. Note that for larger bit-lengths $\sigma \geq 64$ and for $n = 2^{24}$ elements, the memory needed for the OT-based PSI protocol of [51] exceeded the available memory.

Run-Time (Table 5.3) As expected, the lowest run-time for the equal set-size experiments is achieved by the (insecure) naive hashing protocol followed by the server-aided protocol of [36], which has around twice the run-time. In the LAN setting, however, for short bit-length $\sigma = 32$, our OT-Phasing protocol nearly achieves the same run-time as both of these solutions (which are in a different security model). In particular, when computing the intersection for $n = 2^{24}$ elements, our OT-Phasing protocol requires only 3.5 more time than the naive hashing protocol and 2.5 more time than the server-aided protocol. In comparison, for the same parameters, the original OT-based PSI protocol of [51] has a 68 times higher run-time than the naive hashing protocol, and the DH-based ECC protocol of [49] has a four orders of magnitude higher run-time compared to naive hashing.
While the run-time of our OT-Phasing protocol increases with the bit-length of elements, for $\sigma = 128$-bit its run-time is only 15 times higher than the naive hashing protocol, and is still nearly two orders of magnitude better than the DH-based ECC protocol.
Overall, in the LAN setting and for larger sets (e.g., $n = 2^{24}$), the run time of OT-Phasing is 20x better than that of the original OT-based PSI protocol of [51], and 60-278x better than that of the DH-ECC protocol of [49].
When switching to the WAN setting, the run-times of the protocols are all increased by a factor of 2-6. Note that the faster protocols suffer from a greater performance loss (factors of 5 and 6 for 2^{20} elements, for the naive hashing protocol and server-aided protocol) than the slower protocols (factor 3 for the DH-based and our OT-Phasing protocol and 2.5 for the OT-based PSI protocol of [51]). This difference can be explained by the greater impact of the high latency of 97 ms on the run-time of the protocols. The relative performance among the protocols remains similar to the LAN setting.

Communication (Table 5.4) The amount of communication performed during protocol execution is often more limiting than the required computation power, since the latter can be scaled up more easily by using more machines. The naive hashing approach has the lowest communication among all protocols, followed by the server-aided solution of [36]. Among the secure two-party PSI protocols, the DH-based ECC protocol of [49] has the lowest communication. In the setting for $n = 2^{24}$ elements of short bit-length $\sigma = 32$ bit, our OT-Phasing protocol nearly achieves the same complexity as the DH-based ECC protocol, which is due to the use of permutation-based hashing. This is quite surprising, as protocols that use public-key

Setting	LAN				WAN			
Protocol	$n_2 = 2^8$		$n_2 = 2^{12}$		$n_2 = 2^8$		$n_2 = 2^{12}$	
	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$
Naive Hashing(*)	464	7,739	466	7,836	560	2,775	562	2,797
Server-Aided(*) [36]	680	8,935	696	8,965	629	2,923	731	2,951
DH-based ECC [49]	$4.2 \cdot 10^5$	$6.8 \cdot 10^6$	$4.2 \cdot 10^5$	$6.8 \cdot 10^6$	$1.1 \cdot 10^5$	$1.7 \cdot 10^6$	$1.1 \cdot 10^5$	$1.7 \cdot 10^6$
OT-Phasing								
Bit-length $\sigma = 32$	906	9,465	2,949	12,634	2,139	4,780	3,143	11,399
Bit-length $\sigma = 64$	1,506	15,789	6,146	22,368	3,349	6,879	3,923	20,345
Bit-length $\sigma = 128$	1,942	21,843	7,291	31,932	3,352	7,999	4,391	23,209

Table 5.5: Run-time in ms for PSI protocols with $n_2 \ll n_1$ elements. (Protocols with (*) are in a different security model.)

Protocol	$n_2 = 2^8$			$n_2 = 2^{12}$			Asymptotic [bit]
	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	
Naive Hashing(*)	0.5	8.5	144.0	0.5	9.0	152.0	$n_1\ell$
Server-Aided(*) [36]	1.0	16.0	256.0	1.1	16.1	256.1	$(n_1 + n_2 + \lvert X \cap Y \rvert)\kappa$
DH-based ECC [49]	2.5	40.5	656.0	2.7	41.1	664.1	$(n_1 + n_2)\varphi + n_1\ell$
OT-Phasing							
Bit-length $\sigma = 32$	1.1	18.1	288.1	2.0	18.9	320.9	$4.8n_2\kappa(\lceil\frac{\sigma - \lfloor\log_2(2.4n_2)\rfloor}{8}\rceil) + 2n_1\ell$
Bit-length $\sigma = 64$	1.1	18.1	288.1	3.2	20.1	322.1	$4.8n_2\kappa(\lceil\frac{\sigma - \lfloor\log_2(2.4n_2)\rfloor}{8}\rceil) + 2n_1\ell$
Bit-length $\sigma = 128$	1.1	18.2	288.2	3.5	20.4	322.7	$4.8n_2\kappa(\lceil\frac{\sigma - \lfloor\log_2(2.4n_2)\rfloor}{8}\rceil) + 2n_1\ell$

Table 5.6: Communication in MB for special purpose PSI protocols with $n_2 \ll n_1$ elements. $\ell = \sigma + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n_2$ for the TTP-based protocol. (Protocols with (*) are in a different security model.)

cryptography, in particular elliptic curves, were believed to have much lower communication complexity than protocols based on other cryptographic techniques.
In comparison to the original OT-based PSI protocol of [51], OT-Phasing reduces the communication by factor 2.5 - 4. We can also observe that OT-Phasing reduces the impact when performing PSI on elements of longer bit-length. In particular, OT-Phasing has lower communication overhead than the original OT-based PSI protocol for all combinations of elements and bit-lengths. In fact, it even has a lower communication for $\sigma = 128$ than the original OT-based PSI protocol has for $\sigma = 32$.

Experiments with Different Input Sizes

For examining the setting where the two parties have different input sizes, we set $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n_2 \in \{2^8, 2^{12}\}$ and run the protocols on all combinations such that $n_2 \ll n_1$. Note that we excluded the original OT-based PSI protocol of [51] from the comparison, since the bin size max_β becomes large when $\beta \ll n$ and the memory requirement when padding all bins to max_β elements quickly exceeded the available memory. In this setting, unlike the equal input sizes experiments, we use $h = 2$ hash functions instead of $h = 3$, since this results in less total computation and communication. Since we use $h = 2$ hash functions, we also increase the number of bins from $1.2n_2$ to $2.4n_2$. Furthermore, we do not use a stash for our OT-Phasing protocol with different input sizes, since the stash would greatly increase the overall communication. However, not using a stash reveals some information on $\mathcal{B}$'s set.

Run-Time (Table 5.5) Similar to the results for equal set sizes, the naive hashing protocol is the fastest protocol for all parameters. The server-aided protocol of [36] is the second fastest protocol but it scales better than the naive hashing protocol for increasing number of elements. The best scaling protocol is our OT-Phasing protocol. It achieves the same performance as

the server-aided protocol for $n_2 = 2^8$, $n_1 = 2^{24}$ with short bit-length $\sigma = 32$. For $n_1 = 2^{24}$ its run-time is at most twice that of the server-aided protocol in both network settings.
When switching to the WAN setting, the run-times of all protocols are increased by a factor 4-6 while the relative performance between the protocols remains similar, analogously to the equal set size experiments.

Communication (Table 5.6) As expected, the naive hashing solution again has the lowest communication overhead. Surprisingly, our OT-Phasing protocol achieves nearly the same communication as the server-aided protocol of [36] and has only two times the communication of the naive hashing protocol for all bit-lengths. Furthermore, our OT-Phasing protocol requires a factor of 2-3 less communication than the DH-based ECC protocol of [49] for nearly all parameters. The low communication of our OT-Phasing protocol for unequal set sizes is due to the low number of OTs performed.

Chapter 6

Conclusion

The report described new protocols for secure multi-party computation. The main goal of the work that was performed was to address the needs of applications, by improving the performance of protocols which fit the requirements of the application scenarios that were described in Deliverable D11.2 of this project. The protocols that were presented in this report have been published in multiple research papers at top-tier conferences.
The results that was presented in this deliverable can be categorized into several categories:

- Chapter 2 described improved protocols for generic secure multi-party computation, namely protocols that can be used for securely computing any function.
- Chapter 3 described improved building blocks for constructing secure protocols. Namely better constructions of oblivious transfer, token-aided computation, secure arithmetic operations, and zero-knowledge proofs, which are tools that are used by secure computation protocols.
- Chapter 4 described new protocols for the specific problem of search on encrypted data. In that setting the input (an encrypted database) is of a huge size, and therefore generic protocols are not sufficiently efficient.
- Chapter 5 described improved protocols for the specific problem of computing the intersection of two private sets. This is a problem of high interest and therefore we designed protocols that are tailored for solving this problem and are significantly more efficient that applying generic protocols to this problems.

This deliverable is a preliminary report that was written after the second year of the project. We are currently working on designing further improved protocols, based on our analysis experimentation of the current protocols.

Chapter 7

List of Abbreviations

2PC	Two party computation
ABY	Arithmetic-Boolean-Yao
AES	Advanced encryption standard
BMR	the Beaver-Micali-Rogaway protocol
DH	Diffie-Hellman
EC	European Commission
ECC	Elliptic curve cryptography
FHE	Fully homomorphic encryption
GC	Garbled circuit
GMW	the Goldreich-Micali-Wigderson protocol
GRR	Garbled row reduction
IR	Ireland
MPC	Multi-party computation
OT	Oblivious transfer
SCS	Sort-compare-shuffle
SPDZ	the Damgard-Pastro-Smart-Zakarias protocol
PRF	Pseudo random function
PSI	Private set intersection
VA	Virginia
ZK	Zero knowledge

Bibliography

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, 2006.

[2] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science (FOCS'10)*, pages 787–796. IEEE, 2010.

[3] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Computer and Communications Security (CCS'13)*, pages 535–548. ACM, 2013.

[4] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *Advances in Cryptology – EUROCRYPT'15*, volume 9056 of *LNCS*, pages 673–701. Springer, 2015. Full version: http://eprint.iacr.org/2015/061.

[5] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security*, pages 535–548. ACM, 2013.

[6] M. J. Atallah, M. Bykova, J. Li, K. B. Frikken, and M. Topkara. Private collaborative forecasting and benchmarking. In *Workshop on Privacy in the Electronic Society (WPES'04)*, pages 103–114. ACM, 2004.

[7] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

[8] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Symposium on Theory of Computing (STOC'96)*, pages 479–488. ACM, 1996.

[9] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In Harriet Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.

[10] M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Symposium on Security and Privacy (S&P'13)*, pages 478–492. IEEE, 2013.

[11] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security*, pages 784–796, 2012. Full version at http://eprint.iacr.org/2012/265.

[12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer-Verlag Berlin Heidelberg, 2011.

[13] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, 2009.

[14] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security (ESORICS'08)*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.

[15] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 184–198, 2014.

[16] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[17] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2012.

[18] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Garay and Gennaro [27], pages 513–530.

[19] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, pages 40–58, 2015.

[20] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 2012*, volume 7485 of *LNCS*, pages 241–263. Springer, 2012.

[21] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.

[22] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [57], pages 643–662.

[23] D. Demmler, T. Schneider, and M. Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *USENIX Security Symposium (USENIX Security'14)*, pages 893–908. USENIX, 2014.

[24] D. Demmler, T. Schneider, and M. Zohner. ABY - a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security (NDSS'15)*. The Internet Society, 2015.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[26] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. *EUROCRYPT*, 2015:598, 2015.

[27] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*. Springer, 2014.

[28] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Symposium on Theory of Computing (STOC'87)*, pages 218–229. ACM, 1987.

[29] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *SIGPLAN Not.*, 29(6):61–72, June 1994.

[30] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *Computer and Communications Security (CCS'10)*, pages 451–462. ACM, 2010.

[31] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed Security Symposium (NDSS'12)*. The Internet Society, 2012.

[32] IETF. The Transport Layer Security (TLS) Protocol Version 1.2. Technical report, Internet Engineering Task Force (IETF), 2008.

[33] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *ACM Symposium on Theory of Computing (STOC'89)*, pages 44–61. ACM, 1989.

[34] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[35] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM Conference on Computer and Communications Security*, pages 955–966, 2013.

[36] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. In *Financial Cryptography and Data Security (FC'14)*, volume 8437 of *LNCS*, pages 195–215. Springer, 2014.

[37] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM CCS '13*, pages 549–560. ACM, 2013.

[38] F. Kerschbaum, T. Schneider, and A. Schröpfer. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security (ACNS'14)*, volume 8479 of *LNCS*, pages 566–584. Springer, 2014. Extended abstract published in NDSS'13.

[39] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[40] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 275–286. ACM, 2014.

[41] Joe Kilian. Founding cryptography on oblivious transfer. In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31. ACM, 1988.

[42] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology – CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[43] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology And Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.

[44] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[45] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In Garay and Gennaro [27], pages 440–457.

[46] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *LNCS*, pages 1–17. Springer, 2013.

[47] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [27], pages 476–494.

[48] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*, pages 287–302. USENIX, 2004.

[49] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Symposium on Security and Privacy (S&P'86)*, pages 134–137. IEEE, 1986.

[50] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [57], pages 681–700.

[51] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium*, pages 797–812. USENIX, 2014.

[52] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530. USENIX Association, 2015.

[53] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 250–267, Berlin, Heidelberg, 2009. Springer-Verlag.

[54] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, 2009.

[55] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[56] P. Pullonen, D. Bogdanov, and T. Schneider. The design and implementation of a two-party protocol suite for SHAREMIND 3. Technical report, CYBERNETICA Institute of Information Security, 2012. T-4-17.

[57] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *LNCS*. Springer, 2012.

[58] T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, volume 7859 of *LNCS*, pages 275–292. Springer, 2013.

[59] A. C. Yao. Protocols for secure computations. In *FOCS'82*, pages 160–164. IEEE, 1982.

[60] A. C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

[61] Andrew Chi-Chih Yao. Protocols for secure computations. In *Proceedings FOCS 1982*, pages 160–164. IEEE Computer Society, 1982.

[62] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2015.

[63] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE, 2006.