D1.1 State of the Art Analysis of MPC Techniques and Frameworks

Peter S. Nordholt (ALX), Nikolaj Volgushev (ALX), Prastudy Fauzi (AU), Claudio Orlandi (AU), Peter Scholl (AU), Mark Simkin (AU), Meilof Veeningen (PHI), Niek Bouman (TUE), Berry Schoenmakers (TUE)

The project SODA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

Project Information

Scalable Oblivious Data Analytics

Project number:	731583
Strategic objective:	H2020-ICT-2016-1
Starting date:	2017-01-01
Ending date:	2019-12-31
Website:	https://www.soda-project.eu/

Document Information

Title: D1.1 State of the Art Analysis of MPC Techniques and Frameworks

ID:	D1.1	Type:	R	Dissemination level:	PU
		Month:	M9	Release date:	September 29, 2017

Contributors, Editor & Reviewer Information

Contributors (person/partner)	Peter S. Nordholt (ALX) Nikolaj Volgushev (ALX) Prastudy Fauzi (AU) Claudio Orlandi (AU) Peter Scholl (AU) Mark Simkin (AU) Meilof Veeningen (PHI) Niek Bouman (TUE) Berry Schoenmakers (TUE)
Editor (person/partner)	Prastudy Fauzi (AU), Claudio Orlandi (AU)
Reviewer (person/partner)	Meilof Veeningen (PHI), Niek Bouman (TUE)

Release History

Release	Date issued	Release description / changes made
1.0	September 29, 2017	First release to EU

SODA Consortium

Full Name	Abbreviated Name	Country
Philips Electronics Nederland B.V.	PHI	Netherlands
Alexandra Institute	ALX	Denmark
Aarhus University	AU	Denmark
Göttingen University	GU	Germany
Eindhoven University of Technology	TUE	Netherlands

Table 1: Consortium Members

Executive Summary

Secure multi-party computation (MPC) can be seen as a tool that allows a number of distributed entities to jointly compute a functionality on their private data securely, even if some of the parties involved are dishonest. The fact that it can be used to perform *any* functionality means that there are a myriad of interesting scenarios, including in data analytics, where one can use MPC. The main question is how practical the MPC-based solutions are, especially in the *big* data setting, and whether or not this can be improved, without compromising the desired security properties.

A function is commonly modelled as a Boolean or arithmetic circuits, but some recent work have investigated the use of fixed/floating point arithmetic and oblivious RAM. As we will discuss in Section 2, two-party computation is done efficiently using Yao's garbled circuits or GMW-style secret sharing. The general consensus is that Yao is preferred if we want to minimize rounds or if the network latency is high, while GMW is preferred if we want to minimize bandwidth, or if we only care about online computation. However, recent results have indicated that this is far from clear cut.

If we have three or more parties, we can talk about the notion of an honest majority. For three parties with at most one corruption, there are many efficient constructions, the most notable of which are Sharemind and Viff, which use secret sharing over a large field. But there has also been some recent constructions which use secret sharing over $\mathbb{F}_2$.

For more than three parties and/or a dishonest majority, most of the efficient constructions also use secret sharing, with active security guaranteed using information-theoretic MACs. The bulk of the effort is done in the preprocessing phase in the form of generation of many multiplication triples. The most efficient results to date are SPDZ and MASCOT.

RAM programs have the advantage that it models the workings of a real world program, and its size is not dependent on the function to be computed or the size of the input. However, as will be discussed in Section 4, it is a challenge to make RAM access *oblivious* and MPC-friendly. Current ORAM constructions vary in the amount of storage, online operations, and bandwidth required.

To make MPC practical, it is important to have efficient MPC-based implementations of common abstract data types, which result in *oblivious data structures*. The main considerations here include what is leaked in each data structure, how data is partitioned, support for concurrency and what MPC techniques are compatible.

As we will see in Section 5, MPC alone does not guarantee privacy and anonymity, since the output itself might leak sensitive information. Therefore, we need techniques in *differential privacy*, which aims to provide accurate computation on database elements without giving information about individual database entries. Combining differential privacy and MPC techniques seems to be the way forward to perform secure big data analytics. However, this is far from straightforward to achieve, and is among some of the research directions we aim to pursue in WP1.

In Section 6 we claim that MPC is not just a theoretical concept by showing that there exist several general-purpose software frameworks for MPC, i.e., frameworks that can evaluate any computable function. Some of the best alternatives include VIFF, Sharemind, Bristol-SPDZ, Wysteria, ABY, OblivVM, Obliv-C and FRESCO. These frameworks have been used in practice, and choosing the best framework to use depends on several factors, such as preferred programming language, what function you wish to implement, and how you want to model the function.

About this Document

Role of the Deliverable

This deliverable contains state of the art analysis in secure multi-party computation (MPC), which includes a survey of both theoretical results and practical implementations. The goal of this deliverable is to provide an overview of the best available techniques in the state-of-the-art as opposed to a complete historic account of all known techniques. It is part of Work Package 1 of the SODA project.

Relationship to Other SODA Deliverables

This state-of-the-art deliverable will serve as a basis for further research on secure multi-party computation, which will be presented in deliverables D1.2 and D1.3. Deliverable D2.1 also contains state of the art analysis but focuses on privacy-preserving data mining protocols, including a survey on the use of multi-party protocols in data mining. Sections 5 and 6 relate both to this work package and to tasks Tasks 3.1 and 4.1, respectively; they are included in this deliverable since these respective work packages do not have their own state-of-the-art analysis.

Structure of this Document

The first section gives an introduction to MPC, including its motivation and some essential notions. The second section gives an overview of two-party computation up to some of the latest results. The third section discusses secure computation in the case of three or more parties, as well as a discussion on verifiable computation. The fourth section discusses data-oblivious MPC, in particular oblivious RAM and oblivious data structures. The fifth section takes a look at the different notions in differential privacy and its use in MPC. The sixth section gives an overview of practical implementations of MPC. Finally, the seventh section concludes the state of the art document with the main points of the previous sections and the type of MPC-related challenges that we aim to overcome during the SODA project.

Table of Contents

1 Introduction

Secure multi-party computation (MPC) is a method where several parties in a distributed setting can work together to securely evaluate a function of their combined inputs, while keeping their individual inputs private. Security in this case means that the desired properties of the protocol (such as privacy of inputs) must hold even if an entity (such as a party participating in the MPC) tries to attack the protocol

There are various scenarios where MPC can be useful, the following are some examples as an illustration.

- *Secure electronic auctions*. In a sealed auction, a number of bidders would like to secretly bid on an item, such that the highest bidder will win the auction. A bidder aims to win the auction without bidding too much. In the electronic version, the bidder would additionally like its bid to be kept secret, during and after the auction.
- *Private data analysis*. A number of hospitals want to compute an operation on their private datasets (e.g., to get a list of patients that have symptoms of a certain disease), such that the correct result is obtained without revealing anything about the sensitive patient data involved in the analysis.

1.1 Computational Models of MPC

The first step to compute a function securely is to determine how we want the function to be represented.

1. Boolean: a function is modelled as a Boolean circuit. Inputs to a function are Boolean vectors, and the output is also a Boolean vector. The circuit consists of two basic gates, i.e., XOR and AND. Individual gates are easy to evaluate, but the circuit itself can be large and have large depth.
2. Arithmetic: a function is modelled as an arithmetic circuit. Inputs to a function as well as the output of the function are integers or integers in $\mathbb{Z}_p$. The circuit consists of two basic operations, i.e., add and multiply. An arithmetic circuit in general has much smaller size and depth compared to a Boolean circuit that computes the same function, but an arithmetic operation is more costly to implement.
3. Fixed/floating point: inputs to a function can be fixed or floating point numbers, in contrast to Boolean vectors or integers in $\mathbb{Z}_p$.
4. RAM: a function is modelled as a RAM program, allowing for memory accesses as well as arithmetic operations.

The above models can accommodate computation of any functionality in general. There are techniques to compute specific functionalities that do not fall into these computational models, but rather make use of the properties of the desired functionality to get efficient solutions. For example, to get an intersection of two private sets, there are many well-known *private set intersection* protocols [154, 115]. In this document we will focus on computation using the four models described above, and do not cover protocols for specific functionalities.

1.2 Security Definition

To define security of a secure function evaluation, we first must agree on what are the desired properties of a multi-party computation. Below we mention the most natural ones.

- Privacy: No party can learn anything about other parties' inputs, except that which can be learned from the output of the function. This of course does not rule out the possibility that other parties' inputs can be deduced from the output. For instance, if party P_1 with input x and party P_2 with input y want to compute their average $f(x,y) = \frac{x+y}{2}$, it is easy for party P_1 to deduce y from his value x and the function's output $f(x,y)$. While it seems unavoidable, in Section 5 we will review techniques to address this concern.
- Correctness: The protocol must compute the correct functionality.
- Independence of inputs: No party can construct their input as a function of other parties' inputs. This means that the protocol must be constructed in such a way that no party can take advantage of order of computation.
- Fairness: Either all parties get the output, or no party does.

1.3 Security Models of MPC

Security of MPC can also be seen from the behaviour of an adversarial party.

- Passive security: The adversary is honest (i.e., performs the protocol as specified), but curious about the inputs of other parties. In this model, we focus on achieving privacy of parties' inputs.
- Active security: The adversary can behave maliciously, for instance by performing an incorrect step in the protocol or by aborting mid-way. In this model, all the desired properties of MPC need to be achieved.
- Covert security: The adversary can behave maliciously, but will be detected with some probability ε.
- Rational: The adversary is rational in a game-theoretic sense, meaning that it operates with a goal of maximizing a specific utility function.

1.3.1 Real vs. Ideal World Paradigm

Let **F** be the functionality that we want to be computed. Security is realized by comparing the real and ideal world execution of the protocol, as follows.

- In the real world, parties execute a protocol to compute **F** in the presence of an adversary. The adversary can be static or dynamic. A *static* adversary corrupts certain parties before the protocol begins, but this choice is fixed until the end of the protocol execution. An *adaptive* adversary can choose which parties to corrupt during the protocol execution.
- In the ideal world, parties give their inputs to a trusted party who in turn computes the functionality **F** and returns the correct output to the corresponding parties.

Using this definition, an MPC protocol is secure if there exists a simulator that emulates output in an ideal world execution such that no adversary can tell the output of the simulator and the real world execution apart with more than negligible probability. If the distribution of the output of simulator and the real world execution are *information theoretically* indistinguishable we talk of *unconditional security* (which can in turn ben *perfect* or *statistical*), which means that security holds even against adversaries with unbounded computing power. If, however, the distribution of the outputs are only *computationally* indistinguishable then we talk about *computational* security i.e., the protocols are secure under the (reasonable) assumption that the adversary does not have the necessary computing power to break the cryptographic assumptions on which the protocol is built upon.

1.4 The Preprocessing Model

A key step towards making protocols more practical is to isolate the costly parts of the protocol into an *offline* or *preprocessing phase* that is independent of the parties' inputs and can be done in advance. The actual computation is then performed in an *online phase* that is much more efficient.

- In the *offline phase*, parties will perform a *preprocessing* protocol before they get the input to the secure function to be computed. This is also known as the *preprocessing phase*.
- In the *online phase*, parties use the values computed in the offline phase along with the input to the function to be evaluated to complete the secure function evaluation.

The benefits of this are twofold. Firstly, the latency experienced by a user after providing their input is much lower than if they have to wait for the entire computation (this is particularly important in server-based applications, when there may be periodic requests from users, but preprocessing can be performed in between). Secondly, creating more modularity in protocols makes it easier to focus on the least efficient aspects and design new, improved protocols for the preprocessing.

In many real-life applications, the offline phase is allowed to be long, but the online phase needs to be as short as possible. For example, in a secure auction it is fine for the setup of the auction to take hours (or even days), but the live bidding process itself should have as little latency as possible, and the output should be generated withing seconds or minutes.

2 Two-Party Computation

In this section we focus on the case of secure functionalities between two parties, party P_1 with input x and party P_2 with input y. At the end of the protocol, P_1 should get a function $f_1(x,y)$ while P_2 gets $f_2(x,y)$. In most cases, we will have $f_1 = f_2 = f$, in which case we have a symmetric function evaluation.

In the ideal world, P_1 and P_2 send their outputs to a trusted party which returns the correct result if none of the parties abort. A two-party protocol is secure if it is indistinguishable to the ideal world execution, but without any trust assumptions. Cleve [44] showed that it is impossible to achieve fairness in general. In light of this, we usually do not focus on achieving fairness when constructing a secure two-party protocol. (Surprisingly, it is actually possible to achieve fairness for a limited class of non-trivial functionalities even in the two-party setting as demonstrated by the breakthrough result of [91]).

2.1 Tools for Two-Party Computation

2.1.1 Oblivious Transfer

A useful tool in two-party computation is *oblivious transfer* (OT), specifically 1-out-of-2 OT. In a 1-out-of-2 OT protocol, a sender S has two elements (x_0,x_1) and a receiver R has a query bit $b \in \{0,1\}$. At the end of the protocol, R receives x_b but knows nothing about x_{1-b}, while the sender does not get any knowledge about receiver's query bit b. OT is usually implemented using public key cryptography. The most efficient known protocol that is secure against malicious adversaries uses the PVW cryptosystem [147]. The notion of OT can be extended to the case of 1-out-of-n OT_l^k, where the sender has a set of k n-tuples $(x_{i1},x_{i2},\ldots,x_{in})_{i=1}^k$ each of size l bits, and the receiver has k query values each in $[1,n]$.

There is a variant of OT called *random OT*, where R does not get a query bit, but instead gets (b,x_b) at the same time. Random OT can be used to perform most of the work of OT to the offline phase, with a very cheap online phase, as follows.

Input: The sender S initially has two elements (x_0,x_1) and the receiver R has a query bit $b \in \{0,1\}$.

Offline: S chooses random elements (y_0,y_1) and performs random OT, with R receiving (c,y_c) back.

Online: R sends a bit $d = b \oplus c$ to S. In response, S sends back (z_0,z_1), where

$$\begin{aligned} z_0 &= x_0 \oplus y_d \\ z_1 &= x_1 \oplus y_{1-d} \ . \end{aligned}$$

R can now obtain $x_b = z_b \oplus y_c$.

The protocol works since if $b = 0$ then R knows $y_c = y_d$, else R knows $y_c = y_{1-d}$ where c is a random bit. Since R does not know y_{1-c} which is sampled uniformly at random, he does not know anything about x_{1-b} from z_{1-b}. Moreover, since S does not know c, he also does not know $b = c \oplus d$. The public key operations are done in the offline phase, hence the online phase only consists of XOR operations and communicating a small number of bits.

2.1.2 OT Extension

As public key operations are expensive, OT is commonly a bottleneck for two-party computations. Hence to gain efficiency it is important to use as few OT calls as possible. Another alternative is to implement a large number of OT instances using only a small number of base OTs, using a technique called *OT extension* [99].

Warm-up: Length Extension. We first show how to extend a single oblivious transfer on strings of length k, for security parameter k, to an OT on strings of an arbitrary length, n. Let $G : \{0,1\}^k :\rightarrow \{0,1\}^n$ be a pseudorandom generator. The two parties run a random OT on length k strings, where the sender uses two random strings (r_0, r_1), and the receiver learns the bit b and the string r_b. The sender takes the input strings $x_0, x_1 \in \{0,1\}^n$, then sends $d_0 = x_0 \oplus G(r_0)$ and $d_1 = x_1 \oplus G(r_1)$ to the receiver. The receiver then uses r_b to compute $x_b = d_b \oplus G(r_b)$.

This protocol allows OT_n^k to be performed at the same cost as OT_k^k, plus a few PRG operations.

OT Extension. The basic construct of OT extension is as follows. Let H be a hash function.

Input: The sender S initially has n pairs of m-bit messages $(x_{i,0}, x_{i,1})_{i=1}^n$, while the receiver has n query bits $\mathbf{b} = (b_1, b_2, \ldots, b_n)$.

Protocol: S and R proceed as follows.

1. S generates a random k-bit vector $\mathbf{s}$, while R generates a random $n \times k$ matrix of bits $M = (M^1 | M^2 | \cdots | M^k)$.
2. S and R run OT_k^n but with the roles reversed: S uses $\mathbf{s}$ as the selection bits for k OTs on each column of M, where R's j-th pair of inputs is $(m_0, m_1) = (M^j, M^j \oplus \mathbf{b})$. After this step, S will get a matrix Q with columns $(Q^1, \cdots, Q^n)$, where $Q^j = M^j \oplus s_j \cdot \mathbf{b}$.
   Notice that the rows of Q and M satisfy $Q_i = M_i \oplus b_i \cdot \mathbf{s}$.
3. For each row of Q, S sends $(y_{i,0}, y_{i,1})$, where

$$\begin{aligned} y_{i,0} &= x_{i,0} \oplus H(i, Q_i) \\ y_{i,1} &= x_{i,1} \oplus H(i, Q_i \oplus \mathbf{s}) \ . \end{aligned}$$

4. R can now compute $x_{i,b_i} = y_{i,b_i} \oplus H(i, M_i)$.

Using the length extension described previously to implement OT_k^n, n OT instances can be done using only k base OTs (here $k \ll n$) and symmetric key operations. Since symmetric key operations are much faster than public key operations, this is obviously an improvement compared to using regular OT n times. This construction is due to Ishai et al. from 2003 [99], and was later optimized by Asharov et al. [9].

2.1.3 Garbled Circuits

Yao's garbled circuit [179] technique enables P_2 to evaluate a gate chosen by P_1, such that P_1 does not know P_2's input and P_2 gets the result without knowing the gate which was evaluated. Let a gate $G : \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$ have input wire u, v and output wire w. To garble it, P_1 first chooses uniformly at random two keys k_u^0, k_u^1 for wire u, and similarly two keys k_v^0, k_v^1 for wire v and two keys

k_w^0, k_w^1 for wire w. Then, using a symmetric encryption E (e.g., AES encryption), P_1 computes the 4 values $E_{k_u^a}(E_{k_v^b}(k_w^{G(a,b)}))$. P_1 then permutes the 4 ciphertexts and sends them to party P_2. For example, Table 2 and Table 3 shows the process for $G(a,b) = a \oplus b$.

u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^1
k_u^1	k_v^0	k_w^1
k_u^1	k_v^1	k_w^0

Table 2: Associated keys for an XOR gate

Garbled XOR
$E_{k_u^1}(E_{k_v^1}(k_w^0))$
$E_{k_u^0}(E_{k_v^1}(k_w^1))$
$E_{k_u^0}(E_{k_v^0}(k_w^0))$
$E_{k_u^1}(E_{k_v^0}(k_w^1))$

Table 3: Garbled table for an XOR gate

P_2 then gets the right keys k_u^a, k_v^b for his input a, b, and decrypts each of the 4 ciphertexts looking for $k_w^{G(a,b)}$. The correct keys must be formatted in such a way to distinguish them from an incorrect decryption (e.g., by adding some leading 0s or adding a permutation bit to the keys and the ciphertexts). Here there needs to be a mechanism to guarantee that for each input wire, P_2 can only get one key for each pair $(k_u^0, k_u^1), (k_v^0, k_v^1)$ depending on his input (in 2PC this is typically done using OT as introduced in the previous subsection, but in other applications of garbled circuits other mechanisms can be used). In this way, P_2 gets $k_w^{G(a,b)}$ but since the keys are random, nothing else is revealed.

This can be generalized to get a garbled circuit, where each gate is garbled as in the above, and the output key pair (k_w^0, k_w^1) can be used as input to following gates. Additionally, the two keys corresponding to the output wire of the whole circuit is not garbled, i.e., just 0 and 1.

2.1.4 Garbled Circuit Optimizations

There are many optimizations of this general construction of garbled circuits. Here we discuss some of the well known techniques.

- *Free XOR.*

  The Free XOR idea of [117] is to have keys involved in XOR have the same global difference δ which is chosen uniformly at random. Then for an XOR gate, by setting

$$\begin{aligned} k_w^0 &= k_u^0 \oplus k_w^0 \\ k_u^1 &= k_u^0 \oplus \delta \\ k_v^1 &= k_v^0 \oplus \delta \\ k_w^1 &= k_w^0 \oplus \delta \ , \end{aligned}$$

we get that $k_w^0 = k_u^0 \oplus k_w^0 = k_u^1 \oplus k_w^1$ and $k_w^1 = k_u^0 \oplus k_w^1 = k_u^1 \oplus k_w^0$.

Essentially this will make evaluating XOR gates free, meaning no encryptions or decryptions are needed in computing or evaluating the garbled table. This will result in huge efficiency improvements for secure evaluation of circuits with many XOR gates.

- *Garbled row reduction.*

  Using the idea of [133] one can "fix" the first entry in a garbled table to always be an encryption of 0, and hence need not be sent. This reduces the number of rows (of ciphertexts) in a garbled table from 4 to 3. The technique, known as 4-to-3 garbled row reduction (GRR), is compatible with Free XOR.

  We can additionally use polynomial interpolation as in [150] to reduce the number of rows in a garbled table from 4 to 2, also known as 4-to-2 GRR. However, this technique is not compatible with Free XOR.

- *FlexOR.*

  To further optimize garbled gates, [116] uses a combination of 4-to-3 GRR, 4-to-2 GRR, and Free XOR. In particular, this technique lets some XOR gates have only 2 wires (and its associated keys) have the same global difference, while the remaining one does not. In this case XOR requires one ciphertext in the garbled table, but when done correctly this flexibility enables more efficient AND gates compared to using Free XOR for every XOR gate.

- *Half gates.*

  Garbled AND gates can be optimized using the idea of [183], which reduces the number of ciphertexts in an AND gate from 4 to 2.

While the most efficient garbled circuit optimizations use strong hardness assumptions (e.g., AES is circular secure), it was shown by Geron et al. [94] that we can use standard assumptions (e.g., AES is a pseudorandom function) without a significant cost in efficiency.

2.1.5 Secret sharing

In a (k,n)-secret sharing scheme, any k out of n parties can work together to recover a secret shared value (using a *reconstruction* protocol), but any $k-1$ parties working together will not know anything about this value. For two-party computation, the relevant case is when $k = n = 2$. In a $(2,2)$-secret sharing scheme, a party alone does not know anything about the secret shared value, but two parties together can recover it. For example, in additive secret-sharing, to secret share a value $a \in \mathbb{Z}_n$, a dealer can generate a random element $a_1 \in \mathbb{Z}_n$ as party P_1's share and set $a_2 = a - a_1$ as party P_2's share. For a variable x additively shared between P_1 and P_2, P_1's share will be denoted by $[x]_1$ while P_2's share will be denoted by $[x]_2$.

A common instantiation is to use the binary field where the shares of a are $[a]_1$ and $[a]_2$ with $a = [a]_1 \oplus [a]_2$. In additive secret sharing, it is trivial to compute, given two secret-sharings, a new secret-sharing of the sum of the two original values, but computing products of secret-shared values is more involved. Using these sum and product primitives, one can use secret sharing to securely compute any arithmetic circuit.

2.1.6 Information-theoretic MAC

A *message authentication code* (MAC) is a value that can be used to confirm a message has been created by a certain party (who knows the *MAC key*), and to detect if a message has been changed.

An information-theoretic MAC scheme consists of three algorithms:

- $k \leftarrow \mathsf{Gen}$: Output a random MAC key k
- $m \leftarrow \mathsf{MAC}(k,x)$: Output a MAC on x under key k
- $0/1 \leftarrow \mathsf{Ver}(k,m,x)$: Check the MAC on x

The security requirement is that the verification algorithm should succeed if and only if m is a valid MAC on x, except with negligible probability. The *information-theoretic* property of the MAC scheme means that security holds even for an unbounded adversary.

A standard instantiation of this is a Carter-Wegman style MAC [176] over a finite field, where $k = (\alpha,\beta) \in \mathbb{F}^2$ is a pair of uniformly random field elements, and $\mathsf{MAC}(k,x) = \alpha \cdot x + \beta$. Verification then simply uses the key to check the linear MAC relation on m and x. Successfully forging a MAC requires guessing α, and so only occurs with probability at most $1/|\mathbb{F}|$, provided the key is only used once.

As we will see, MACs can be used in secret sharing schemes to ensure that corrupted parties cannot lie about their share and cause an incorrect value to be opened during reconstruction.

2.1.7 Commitments

A commitment scheme Com is an algorithm where, given a message m, commitment key ck, and additional value $r \in R$ and outputs $\mathsf{Com}_{\mathsf{ck}}(m;r)$. A commitment scheme has two desired properties:

- *Hiding*: for any m_0, m_1 the distributions $\{\mathsf{Com}_{\mathsf{ck}}(m_0;r)\}_{r\in R}$ and $\{\mathsf{Com}_{\mathsf{ck}}(m_1;r)\}_{r\in R}$ are computationally indistinguishable. Hence given $\mathsf{Com}_{\mathsf{ck}}(m;r)$, a PPT adversary has negligible probability of gaining information about m. It is *perfectly hiding* if commitments to any two messages have the same distribution.
- *Binding*: given a commitment, a PPT adversary has negligible probability of opening the commitment to two different messages.

A popular type of commitments is the Pedersen commitment scheme. These commitments are instantiated in a group G of order p (where the discrete logarithm problem is assumed to be hard), where the commitment key is $\mathsf{ck} = (g,h) \leftarrow_r G^2$, and $\mathsf{Com}_{\mathsf{ck}}(m;r) = g^m h^r$. These commitments are particularly useful in certain MPC protocols because of their *homomorphic property* i.e.,

$$\mathsf{Com}_{\mathsf{ck}}(m_1;r_1) * \mathsf{Com}_{\mathsf{ck}}(m_2;r_2) = \mathsf{Com}_{\mathsf{ck}}(m_1+m_2;r_1+r_2) \ .$$

When this property is not required much faster commitments can be constructed simply by hashing (in the random oracle model) the message together with the randomness i.e., $\mathsf{Com}_{\mathsf{ck}}(m;r) = H_{\mathsf{ck}}(m,r)$.

2.1.8 Zero-knowledge

Let R be a binary relation, and define the language $L = \{x | \exists w : (x,w) \in R\}$, and similarly define $\bar{L} = \{x | x \notin L\}$. We usually assume that $L \in \mathbf{NP}$, in which case one can efficiently verify whether or not $(x,w) \in R$.

In a *zero-knowledge proof*, a prover provides a statement x and tries to convince a verifier that $x \in L$. An zero-knowledge proof has three properties:

- *Completeness*: if $x \in L$, an honest verifier will accept the prover's proof π.
- *Soundness*: if $x \in \bar{L}$, a malicious PPT prover has negligible probability of providing a convincing proof π to an honest verifier.
- *Zero knowledge*: an honest prover's execution can be simulated without knowing the witness, meaning that a verifier cannot learn anything about the witness from the proof. More formally, there exists a simulator S that, given the statement x but not the witness w, outputs $S(x)$ such that $(x, w, S(x))$ and (x, w, π) are indistinguishable, where π is the output of an honest prover.

Commitments and zero-knowledge proofs can be used to transform any protocol secure against semi-honest adversaries into one secure against malicious adversaries using the general approach called the *GMW compiler* [86]. In short, a party's private values in the protocol are "sent" as commitments, and the party provides a zero-knowledge proof that these values are used in the right manner.

Zero-knowledge proofs can also be used to prove knowledge of certain values, such as the Schnorr protocol [157] to prove that one knows the discrete logarithm x of a public value $y = g^x$. Zero-knowledge proofs can be made more efficient if we have a specific computational model in mind. For example, ZKBoo [85] achieves very efficient zero-knowledge proofs for Boolean circuits.

2.2 Two-Party Protocols with Passive Security

2.2.1 Yao

Yao's protocol [95] is based on garbled circuits. The basic construction is for one party, say P_1, to encrypt his function f by transforming it into a circuit C which is then garbled, then send the whole garbled circuit to P_2.

Consider a garbled table for a gate $G(\cdot,\cdot)$. To get the right key $k_w^{G(a,b)}$ for his input a, b, P_2 first gets k_u^a and k_w^b using OT, and decrypts each of the 4 ciphertexts. Since OT guarantees that P_2 only has one key for each input wire, P_2 can only get one of (k_w^0, k_w^1) depending on his choice of (a, b). In this way, party P_2 gets one key for each input wire, and uses them to evaluate the garbled circuit by a sequence of garbled gate evaluations as described in Section 2.1.3.

2.2.2 GMW

The GMW technique is based on secret sharing using the XOR operation. Due to the nature of XOR, computing an XOR gate can be done locally. Meanwhile, computing an AND gate can be done either by using a $(4,1)$-OT in a similar technique as in garbled circuits or by using multiplication triples.

- Assume two parties have a share of $a = [a]_1 \oplus [a]_2$ and $b = [b]_1 \oplus [b]_2$, and want to secret share $c = a \wedge b$. P_1 first generates a random bit x, then creates a table based on all possible values for P_2's shares, as seen in Table 4. P_2 then obtains $y \in \{y_0, y_1, y_2, y_3\}$ using $(4,1)$-OT. In this way P_1 gets a share $[c]_1 = x$ and P_2 gets a share $[c]_2 = y$ of the secret value $c = [c]_1 \oplus [c]_2 = a \wedge b$.
- Multiplication triples are pre-generated values a, b, c, secret-shared between P_1 and P_2, such that $a \cdot b = c$. Given multiplication triples and shares of values x and y, the online cost of computing a share of $z = x \cdot y$ will require the following steps.
  1. The parties compute and exchange $[d]_i = [x]_i \oplus [a]_i$ and $[e]_i = [y]_i \oplus [b]_i$.

$[a]_2$	$[b]_2$	$c = a \wedge b$	y
0	0	$v_0 = [a]_1[b]_1$	$y_0 = v_0 \oplus x$
0	1	$v_1 = [a]_1([b]_1 \oplus 1)$	$y_1 = v_1 \oplus x$
1	0	$v_2 = ([a]_1 \oplus 1)[b]_1$	$y_2 = v_2 \oplus x$
1	1	$v_3 = ([a]_1 \oplus 1)([b]_1 \oplus 1)$	$y_3 = v_3 \oplus x$

Table 4: Values (y_0, y_1, y_2, y_3) needed to compute AND using $(4,1)$-OT.

2. Let $d = [d]_1 + [d]_2$ and $e = [e]_1 + [e]_2$. The parties can now compute their shares of z as

$$[z]_1 \quad = \quad (d \cdot e) \oplus ([a]_1 \cdot e) \oplus ([b]_1 \cdot d) \oplus [c]_1$$

This works since

$$\begin{aligned} [z]_1 \oplus [z]_2 &= (d \cdot e) \oplus (a \cdot e) \oplus (b \cdot d) \oplus c \\ &= (x \oplus a)(y \oplus b) \oplus (a \cdot e) \oplus (b \cdot d) \oplus (a \cdot b) \\ &= (x \oplus a)(y \oplus b) \oplus (a \cdot (y \oplus b)) \oplus (b \cdot (x \oplus a)) \oplus (a \cdot b) \\ &= x \cdot y \\ &= z \ . \end{aligned}$$

Note that the pre-generated multiplication triples can only be used once, then should be thrown away. Computing AND using multiplication triples still involves random OT for each triple, but it can be made very efficient using OT extensions. A more detailed explanation of multiplication triples will be given in Section 3.2.1.

The basic GMW protocol can be optimized for instance by optimizing AND gates or by computing AND gates of the same level in parallel. Using the optimizations by Schneider and Zohner [156], an implementation of GMW can perform better than current implementations of Yao.

2.3 Two-Party Protocols with Active Security

2.3.1 Cut-and-choose

While Yao's protocol is secure in the semi-honest model, it is not secure in the malicious model. For instance, P_1 can construct and incorrect circuit which computes a different function that what was agreed upon. If this circuit computes a function of P_2's input, then this will break both P_2's privacy and correctness.

The cut-and-choose technique transforms a passively secure Yao's protocol into an actively secure one. The basic idea of cut-and-choose is that party P_1 creates s garbled circuits of the same Boolean circuit, then sends them to party P_2. P_2 will ask P_1 to reveal a fraction (e.g., $s/2$) of the circuits, and checks if they are correct. If yes, P_2 will evaluate the remaining circuits using Yao's protocol, and outputs the majority value from these evaluations.

Note that aborting the protocol will not work, since P_1 can then launch a *selective failure attack* where, unless it has input of a specific form, P_2 will not notice that P_1 misbehaves. In this case P_1 learns part of P_2's input based on whether or not the protocol was aborted. Hence to protect privacy,

P_2 cannot abort even if it knows that P_1 misbehaved during the protocol. To guarantee correctness with overwhelming probability, one can use the majority function on the evaluated circuits instead.

In the cut-and-choose protocol of Lindell [122], an incorrect check or two different outputs of Yao's protocol will enable P_2 to get a trapdoor which can be used to learn P_1's input x. If this happens, then P_2 can just compute $f(x,y)$ on its own. If not, then P_2 will compute the majority of the outputs. Hence, P_1 successfully cheats if P_2 chooses only the correct circuits in the check phase, and chooses only the incorrect circuits in the evaluation phase. Since P_1 created s garbled circuits, this happens with probability 2^{-s}. Hence to get security 2^{-40} against a malicious adversary, it is sufficient to create 40 copies of the garbled circuit.

2.3.2 LEGO cut-and-choose

The LEGO (Large Efficient Garbled-circuit Optimization) cut-and-choose method [138] builds upon the above idea, but does cut-and-choose on individual gates instead of the entire circuit. Instead of just taking a majority of the individual gates, the correct output is guaranteed by having a fault-tolerant circuit. More precisely, the unopened gates are put into buckets, which can be combined into a garbled circuit. In this case, the intended function can be computed correctly as long as there are only a small number of faulty gates in each bucket.

In LEGO, a circuit is comprised of garbled NAND gates along with their input and output wires. Input and output wires of the whole circuit can take a value $c \in \{0,1\}$, while internal wires may additionally take a special value $c = 2$. Each wire will have a so-called zero-key K_0 along with a Pedersen commitment $\mathsf{Com}(K_0)$. As a wire w can take a value $c \in \{0,1,2\}$, two additional keys $K_1 = K_0 + \Delta \mod p$ and $K_2 = K_0 + 2\cdot\Delta \mod p$ are defined, such that w taking a value c is associated with the key K_c (here, $\Delta \in \mathbb{Z}_p$ is called the global difference, which should be unknown to P_2).

At the start of the protocol, P_1 knows K_0 and $\mathsf{Com}(K_0)$ for each wire, while P_2 just knows $\mathsf{Com}(K_0)$. During the protocol, P_2 can learn $K_c \in \{K_0, K_1, K_2\}$ to help it evaluate a gate without learning the value c. However, it may instead learn a set of potential keys K.

The garbled NAND gates themselves are comprised of smaller units named bricks.

1. NT brick. Computes the not-two function, where $z = nt(x)$ is 1 iff x is not 2. The potential key size will linearly increase by a factor of 3.

2. Add brick. Computes the addition function. The potential key size will quadratically increase.

3. Shift brick. Makes it possible to connect brick A to brick B when the output wire of A does not have matching zero-keys with the input wire of B.

4. KF brick. Intersects the potential keys with $\{K_0, K_1\}$, filtering to at most two possibilities (one of which is correct).

Note that a garbled NAND gate can be computed with just the NT, Add and Shift bricks (since x NAND $y = nt(x+y)$), but would then make the size of K grow exponentially with the number of NAND gates, which will be too big in practice. A short overview of the different bricks can be seen in Table 5.

The biggest cost of LEGO is in combining the individual gates into the correct garbled circuit. This involves choosing the replication factor ℓ for each NAND gate, which is shown to get κ-bit security when $\ell = O(\kappa/\log|C|)$. Here the output of each (replicated) NAND gate will be computed using the majority function. LEGO outperforms the general cut-and-choose technique if the number of gates is large.

Brick	Functionality	$\lvert \mathsf{K}_{in} \rvert$	$\lvert \mathsf{K}_{out} \rvert$
NT	Not-two, i.e., $z = nt(x)$ is 0 iff $x = 2$	n	$\leq 3n$
Add	Addition, i.e., $z = x + y$	n	$\leq \frac{n(n+1)}{2}$
Shift	Change zero-key for w to zero-key for w'	n	n
KF	Filter the potential keys to size 2	n	2

Table 5: Size of potential keys set in LEGO bricks.

MiniLEGO [74] improved upon the efficiency of LEGO by replacing Pedersen commitments with an XOR-homomorphic commitment scheme based on OT and error-correcting codes, tweaking OT extension to mostly use symmetric cryptography, and replacing LEGO bricks with more standard garbled gates which enable the various garbled circuit optimizations. The entire protocol then only uses public key cryptography in the seed OTs, while everything else uses symmetric cryptography.

More recently, TinyLEGO [73] further improved upon MiniLEGO and introduced interactive garbling, where both parties work together to compute garbled gates. TinyLEGO optimizes how the buckets are constructed, and is more flexible with what fraction of circuits are checked during cut-and-choose.

2.3.3 TinyOT

TinyOT [137] adapts the GMW protocol into an actively secure one by using MACs on all bits to prevent malicious changing of any intermediate values. Let $\delta = [\delta]_1 + [\delta]_2$ be a MAC key, and let $x = [x]_1 + [x]_2$ be a bit. Then we will have the MAC of x as $\delta \cdot x = ([\delta]_1 + [\delta]_2)([x]_1 + [x]_2) = [\gamma]_1 + [\gamma]_2$. As in GMW, the shares $[\gamma]_1, [\gamma]_2$ can be computed using multiplication triples, or directly using OT.

Hence TinyOT has an overhead from GMW in the form of OTs required to compute the MAC shares. Efficiency is gained by using OT extension, and it is secure in the random oracle model. TinyOT has $\Theta(\kappa)$ computation and communication costs.

2.3.4 MiniMAC

MiniMAC [53] is a protocol to securely compute Boolean circuits that is asymptotically more efficient than TinyOT. They create many multiplication triples in a Same Instruction, Multiple Data (SIMD) manner, where the goal is to output vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ (plus joint MACs for active security) such that $x_i \cdot y_i = z_i$. The main idea here is that the shares of these vectors can be transformed into codewords, and the MACs are done on the encoded vectors $E(\mathbf{x}), E(\mathbf{y}), E(\mathbf{z})$. Hence for an adversary to forge $\mathbf{x}'$ such that $\mathsf{MAC}(E(\mathbf{x}')) = \mathsf{MAC}(E(\mathbf{x}))$, he needs to forge a MAC at every place where these codewords differ, i.e., the number of which is the Hamming distance between the codewords. Security is guaranteed by choosing a linear code where one can set a high enough minimum distance between different vectors, such as Reed-Solomon codes. Codes need to be linear to ensure the MAC checks work as in previously described protocols. MiniMAC has $\Theta(\log \kappa)$ computation and communication costs.

2.3.5 Authenticated Garbling

The BMR compiler can turn Yao's passively secure protocol into an actively secure one which is constant round. Wang et al. [173] use this approach and using information theoretic MACs as in TinyOT to get very efficient actively secure two-party computation using so-called authenticated garbled circuits. BMR, the multi-party version of this protocol, will be explained in Section 3.2.5.

2.3.6 MASCOT

MASCOT [109] uses OT extension to get many multiplication triples efficiently, which ultimately improves the pre-processing phase of actively secure two-party protocols. To ensure correctness, some generated triples are checked (and "sacrificed") using cut-and-choose. However, in the malicious setting, a successful check on sacrificed triples can leak information, e.g., using a selective failure attack. To circumvent this, the MASCOT protocol uses privacy amplification techniques which involves taking random subsets of the remaining triples. More details can be seen in Section 3.2.4.

2.4 Comparison

There are two main techniques to securely evaluate a function f between two parties. The first is Yao, where f is written as a Boolean circuit, and the two-party protocol involves techniques for garbled circuits. The second is GMW, where f is generally written as an arithmetic circuit and involves techniques for secret-sharing over a finite field F.

Additionally, there are two main ideas to achieve active security. The first idea is to use cut-and-choose, where the passive protocol is essentially run in parallel a number of times, with the ability to "open" a fraction of them in the middle to check that parties behave honestly. The second idea is to use information-theoretic MACs, where each message sent contains a tag such that if all tags are valid, a party has negligible probability of behaving maliciously.

Yao vs. GMW. There has been major work done in both directions, and there is no clear answer to which technique is better. Computation-wise, there is not much difference between Yao and GMW, since most of the computation involves symmetric operations. In general Yao has a constant number of rounds (regardless of the circuit size) and the protocol can be amortized if the parties compute the same function a number of times, but the amount of communication required to send a garbled circuit is relatively high. In contrast, GMW has several rounds that use a smaller amount of bandwidth in total compared to Yao, but the large number of rounds can be problematic if the network latency is high. If we only care about online costs, GMW has an advantage that most of the computation (and communication) can be done in the preprocessing phase, making the online phase very efficient.

3 Multi-party Computation

3.1 Three Parties, One Corruption

In this subsection we review some recent work on three-party computation. Historically, research focused on the three-party case at a relatively late stage. The seminal results by Yao [178, 179] focused on the two-party case (solving the millionaires problem with a special-purpose protocol and introducing the garbled circuit approach as a general construction). Subsequently, the fundamental results of [86, 17, 39] focused on the multi-party case.

A major distinction between the two-party case and the multi-party case is the possibility of an *honest majority* in the multi-party case, where the number of honest parties is assumed to *exceed* the number of corrupted parties. We get a lot of benefits from honest majorities such as circumventing Cleve's well-known impossibility result for fairness and allowing for perfect security. Perfect security means that there is no margin of error (zero success probability for the adversary) and there is no need for any computational hardness assumption). The three-party case with (at most) one corruption is the simplest multi-party case with an honest majority.

The three-party case with one corruption arises naturally when two parties, who mutually distrust each other, rely on a *trusted (third) party* to achieve certain security objectives. The third party plays a special role in such protocols, hence the set-up is not symmetric. In secure three-party computation, however, the set-up is usually symmetric, such that any of the three parties is potentially corrupted (for passive attacks, even if all three parties are corrupted at the same time, security is maintained as long as the parties do not collude).

Below, we will first discuss some aspects of two concrete frameworks for multi-party computation, focusing on the protocol aspects. In Section 6, technical details about the use of these frameworks will be presented.

3.1.1 Sharemind

Sharemind is the primary example of a system supporting secure three-party computation (see Section 6 for references). Sharemind is optimized to take full advantage of a three-party setting tolerating at most one passive corruption by using so-called "replicated secret sharing." Replicated secret sharing quickly becomes impractical for a larger number of parties, but for three parties it is very efficient. In fact, for three parties replicated secret sharing is much simpler than Shamir's threshold scheme, which is the default for many multi-party computation schemes. Replicated secret sharing works over any finite ring (Shamir's scheme requires a finite field). Therefore, Sharemind natively supports computations modulo 2^{32}, which is much more natural than doing computations modulo some prime close to 2^{32}, say.

Building on a basic protocol for secure multiplication in an asymmetric three-party setting (two parties plus a trusted helper) due to Du and Atallah [65], Sharemind uses secure multiplication protocols for additively shared values over a finite ring. As usual, secure addition is done locally without the need for any interaction. The protocols for advanced operations such as secure comparison are more involved, but optimizations at various levels have been applied to enhance the performance considerably.

The details of the underlying secure protocols are hidden from the secure programmer, who will actually develop the algorithms in a language called SecreC. Many applications have been developed already, especially in the area of data mining.

3.1.2 VIFF

VIFF (Virtual Ideal Functionality Framework, see Section 6 for references) supports multi-party computation, assuming an honest majority. VIFF has a particularly efficient implementation for some of the basic operations (e.g., generating a shared random number) in the three-party case with a passive adversary. To this end, VIFF uses the technique of pseudo-random secret sharing due to Cramer et al. [45], which reduces interaction between the parties at the expense of increased (local) computation for the parties. The work for the parties increases exponentially in the number of parties, and therefore this technique quickly becomes impractical for larger numbers of parties.

However, the protocols for the many other operations (such as secure multiplication) and more advanced operations (such as secure comparison), still require a lot of interaction, even in the three-party case. When the use of pseudo-random secret sharing is disabled, the system supports any number of parties, assuming an honest majority, due to the use of Shamir's secret sharing scheme as the basic building block.

3.1.3 Optimized 3-Party Computation

Following a similar approach to Sharemind and VIFF, record results in the 3-party honest majority case (with abort) are obtained in the following papers: [7] achieves 7 billion AND gates per second in the passive case, and [76, 6] achieves 1 billion AND gates per second in the active case.

Whereas Sharemind and VIFF operate over large fields, supporting integer arithmetic, these record results focus entirely on the boolean case, only supporting boolean circuits. The XOR (sum) of two additively shared bits over $\mathbb{F}_2$ is trivial. For the AND (product) of two additively shared bits over $\mathbb{F}_2$, the basic idea in the passive case is to use a very simple protocol. Given $x = x_1 + x_2 + x_3$ and $y = y_1 + y_2 + y_3$, one computes shares of $z = x * y = z_1 + z_2 + z_3$ by assuming that each party P_i holds shares x_i, x_{i+1} and y_i, y_{i+1}, for $i = 0, 1, 2$ (indices are taken modulo 3), and establishing that afterwards P_i holds share z_i, z_{i+1}. Hence, a basic form of replicated secret sharing is used (as in Sharemind). Computing shares $z_i = (x_i + x_{i+1}) * (y_i + y_{i+1}) + x_i * y_i$ can be arranged such that only three additions and two multiplications are needed, plus sending a single bit from P_i to P_{i+1}, for $i = 0, 1, 2$. Also, these shares are made "fresh' by adding a pseudo-random sharing of 0. In the active case, the protocol becomes slightly more involved and the number of bits sent from P_i to P_{i+1} increases from one to three bits per run of the protocol for AND.

To achieve the record results, the implementation of these basic protocols are optimized by using bit-slicing and related techniques (bit-slicing 128 instances in parallel, and batching 100 of these, for a total of 12800 instances of the protocol).Very fast implementations of AES were thus obtained, achieving approximately 100000 AES computations per second (per core, on a 20 core pair of Intel CPUs).

3.2 Computation with > 3 Parties and Dishonest Majority

When it comes to tolerating a *dishonest majority* of corrupted parties, the situation is much more difficult. In this case unconditional security is impossible to achieve, and even with computational security one must settle for a weaker model without fairness, where a corrupted party may abort the protocol after learning the output, denying the result to honest parties.

3.2.1 Multiplication triples.

The preprocessing model began with Beaver's 'circuit randomisation' technique [14], which allows any arithmetic circuit to be securely computed by randomising the inputs to each multiplication gate, using a *multiplication triple*, which is a triple of secret shared values $[a],[b],[c]$ where a and b are uniformly random (in a finite field) and $c = a \cdot b$. Given this, two secret shared values $[x],[y]$ can be multiplied by publicly reconstructing $d := x - a$ and $e := y - b$ and then computing

$$\begin{aligned}[z] &:= [c] + d \cdot [b] + e \cdot [a] + d \cdot e \\ &= [a \cdot b + (x-a) \cdot b + (y-b) \cdot a + (x-a) \cdot (y-b)] \\ &= [x \cdot y]\end{aligned}$$

Since a and b are uniformly random, revealing d and e does not leak any information on the inputs x, y.

The key advantage of this approach is that any MPC protocol based on linear secret sharing that operates on circuits in a gate-by-gate manner can now be easily recast in the preprocessing model: the preprocessing phase simply consists of creating many multiplication triples, and in the online phase the circuit is securely evaluated using these triples. Note that additions and linear operations can be evaluated locally because the secret sharing scheme is linear. After evaluating the circuit, each party broadcasts their secret shares of the outputs and reconstructs the result. The preprocessing stage is *completely independent* of both the inputs to the function being computed and the function itself (apart from an upper bound on its multiplicative size), and creating triples is typically the bottleneck of a protocol.

3.2.2 Information-Theoretic MACs

When using multiplication triples and a linear secret-sharing scheme with threshold t, we obtain a passively secure MPC protocol that tolerates up to t corruptions. In the dishonest majority setting with up to $n-1$ corruptions, the most natural secret sharing scheme to use is simple additive secret sharing, where $x \in \mathbb{F}$ is shared between n parties by distributing n random shares x_i satisfying $x = \sum_{i=1}^{n} x_i$. Clearly, any $n-1$ shares reveal no information about the secret. However, to achieve *active security*, this is not enough to guarantee correct opening.

We need to ensure that a corrupted party cannot lie about their share and cause an incorrect value to be opened during reconstruction. The main tool for achieving active security in modern, secret sharing-based protocols is information-theoretic MACs. There are two main approaches to using information-theoretic MACs in MPC, namely *pairwise MACs*, introduced in the BDOZ protocol [20], and *global MACs*, as used in the SPDZ protocol [52, 50].

Pairwise MACs. The simplest approach is to tag every secret-shared value with an information-theoretic MAC scheme, such as the one discussed in Section 2.1.6. These are applied to MPC in the BDOZ protocol [20] by modifying the preprocessing stage so that for each shared value $[x]$, party P_i additionally receives a set of MACs on x_i, and MAC keys to verify the other parties' shares, so P_i gets:

$$(x_i, (k_{i,j}, \mathsf{MAC}_{k_{j,i}}(x_i))_{j \neq i})$$

The problem now is that the secret-sharing scheme is no longer linear, as we cannot add together the MACs from two different sets of shares under different MAC keys. To fix this, we modify the key

generation algorithm so that each key $k_{i,j}$ held by P_i is of the form $(\alpha_i, \beta_{i,j}(x_j))$, where α_i is fixed for P_i and $\beta_{i,j}(x_j)$ is fresh for every share x_j. We can now add together the MACs on two shares x_j and y_j, obtaining a correct MAC on $x_j + y_j$ since:

$$(\alpha_i \cdot x_j + \beta_{i,j}(x_j)) + (\alpha_i \cdot y_j + \beta_{i,j}(y_j)) = \alpha_i \cdot (x_j + y_j) + (\beta_{i,j}(x_j) + \beta_{i,j}(y_j))$$

This allows us to use the MACs to verify the openings in a semi-honest protocol based on multiplication triples, resulting in an online phase with only a small overhead on top of the semi-honest protocol.

We remark that, to reduce the overhead of sending and checking MACs, the parties can instead do a *lazy MAC check*, and postpone checking until just before opening the result of the computation. The parties can check all the MACs at once by first sampling a set of public, random field elements (using, say, coin-tossing) and use these to compute a random linear combination of the MACs to be checked. They then check the MAC on the single random combination, which (except with probability $1/\mathbb{F}$) guarantees that all the MACs were correct.

Global Shared MACs. The downside of the BDOZ-style approach is that the amount of preprocessing data is increased by around a factor of n compared with the semi-honest protocol. This can be reduced to a constant with the *SPDZ* approach, which uses a single *secret-shared MAC* on each secret-shared value, instead of MACs on every share. Here, for a secret-shared value $x = \sum_i x_i$, party P_i holds:

$$(\alpha_i, x_i, m_i(x))$$

where $\sum_{i=1}^n m_i(x) = (\sum_{i=1}^n x_i) \cdot (\sum_{i=1}^n \alpha_i)$. As previously, the keys α_i are fixed for every MAC.

We can now think of each α_i as a secret-sharing of the global MAC key $\alpha := \sum_i \alpha_i$, and $m_i(x)$ as a share of the MAC on x. However, now checking the MACs is slightly more difficult, as no single party knows the actual MAC key. After opening the shares of x, the MAC on x can be checked without revealing α as follows:

- Each party P_i commits to $z_i := m_i(x) - x \cdot \alpha_i$
- All parties open their commitments and check that $\sum_{i=1}^n z_i = 0$

As with the BDOZ MACs, this can be optimized for a large batch of MACs to be checked, by only verifying a random linear combination of the MACs.

Converting from Pairwise to Global MACs. In turns out that the two MAC representations described above are related, in the sense that given a shared value x with pairwise BDOZ MACs, the parties can locally obtain SPDZ MACs on x by simply adding together the MACs and $\beta_{i,j}$ components of the keys they hold. This observation was made in the multi-party TinyOT protocol [34], which first sets up pairwise MACs using OT and then combines these into SPDZ MACs for a more efficient online phase.

Complete Online Phase of MPC with Preprocessing. Once we have either of these two MAC representations, it is fairly easy to describe the online phase of an MPC protocol in the preprocessing model, similar to BDOZ or SPDZ. Since both of the authenticated secret-sharing methods described are linear, the parties can perform linear computations by simply computing locally on the shares. We

can also add a public constant c to a shared value $[x]$: P_1 adds c to her share, and each party P_i adjusts their MAC by $c \cdot \alpha_i$ (this is for SPDZ MACs; a similar adjustment can be done for BDOZ MACs).

To multiply two secret-shared values, the parties need a random authenticated multiplication triple from the preprocessing phase. We also need some additional preprocessing data for sharing inputs, in the form of random authenticated masks where only the party providing input knows the value. When outputting a result of the computation the parties must check the MACs using the MAC check procedure described previously, which we denote by Π_{MACCheck}.

The complete protocol follows.

Input: To share an input $x_i \in \mathbb{F}$, party P_i takes a preprocessed mask r_i (such that P_i knows r_i and all parties have shares $[r_i]$). Then do the following:

1. P_i broadcasts $d \leftarrow x_i - r_i$.
2. All parties compute $[x_i] \leftarrow [r_i] + d$.

Add: On input $([x],[y])$, locally compute $[x+y] \leftarrow [x] + [y]$.

Multiply: On input $([x],[y])$, the players do the following:

1. Take one multiplication triple $([a],[b],[c])$, compute $[d] \leftarrow [x] - [a]$, $[e] \leftarrow [y] - [b]$ and open these shares to get d, e respectively.
2. Set $[z] \leftarrow [c] + d \cdot [b] + e \cdot [a] + d \cdot e$

Output: To output a share $[y]$, do the following:

1. Run Π_{MACCheck} with input all unchecked, opened values from multiplications. If it fails, output $\perp$ and abort.
2. Open $[y]$ and check its MAC. If the check fails, output $\perp$ and abort, otherwise accept y as a valid output.

3.2.3 Triple Generation with Homomorphic Encryption

There are two main approaches to generating triples with a dishonest majority, those based on *homomorphic encryption* as in BDOZ [20] and SPDZ [52], and those based on *oblivious transfer* as in the MASCOT protocol [109].

Additively Homomorphic Encryption. In the BDOZ protocol, multiplication triples are created based on the fact that:

$$(\sum_{i=1}^{n} a_i)(\sum_{i=1}^{n} b_i) = \sum_{i=1}^{n} a_i b_i + \sum_{i=1}^{n} \sum_{j \neq i} a_i b_j \tag{1}$$

Each party P_i first samples random shares a_i, b_i, and then can compute the term $a_i b_i$ on its own. To produce shares of the product ab, it suffices for every pair of parties (P_i, P_j) to obtain shares of $a_i b_j$, which is done with the following blueprint.

P_i encrypts a_i with an additively homomorphic encryption scheme and sends the ciphertext $\mathsf{Enc}(a_i)$ to P_j. Using the homomorphic property, P_j then computes and sends back $\mathsf{Enc}(a_i b_j - r)$, where r is a random value. P_i decrypts this to recover a share $s = a_i b_j - r$. The main challenge in this approach is to ensure that both parties send correctly computed ciphertexts. This can be done with a zero-knowledge

proof of plaintext knowledge to show that P_i knows a_i, and a proof of correct multiplication from P_j, to prove that the second ciphertext is well-formed.

Instantiating the additively homomorphic encryption scheme with the required properties can be done using either Paillier based on factoring, or lattice-based schemes with LWE.

Somewhat Homomorphic Encryption. SPDZ uses a *somewhat homomorphic encryption* (SHE) scheme to allow parties to add and multiply secret shared data. Given a set of ciphertexts $\mathsf{c}_1 = \mathsf{Enc}(m_1), \ldots, \mathsf{c}_n = \mathsf{Enc}(m_n)$, SHE allows computation of a ciphertext c such that $\mathsf{Dec}(\mathsf{c}) = f(m_1, \ldots, m_n)$, where f belongs to some class of (multivariate) polynomials. SPDZ requires that f can be a degree two polynomial, which means that any (polynomial) number of ciphertexts can be homomorphically added together, and at most one multiplication can be performed.

To ensure security of the MPC protocol, we need to use SHE with *distributed key generation* and *distributed decryption* protocols. Distributed key generation creates a single public key known to all parties, and a secret key for which each party knows only an additive sharing; distributed decryption then allows the parties to agree to publicly decrypt a ciphertext known by everyone using their secret key shares. Once the keys are set up, whenever a party broadcasts a ciphertext, they must provide a zero-knowledge proof of plaintext knowledge to guarantee this is well-formed.

To create a multiplication triple, each party P_i generates random shares a_i, b_i and computes encryptions of these. All parties broadcast their ciphertexts and sum them up with homomorphic addition to obtain encryptions of $a := a_1 + \cdots + a_n$ and $b := b_1 + \cdots + b_n$. Next, they use homomorphic multiplication to get an encryption of $c := a \cdot b$ and then distributed decryption to obtain a sharing of c. A similar procedure can be used to authenticate the shares, by multiplying by a ciphertext that encrypts the global MAC key α.

One of the main inefficiencies of the BDOZ protocol [20] is the requirement for zero knowledge proofs of correct ciphertext multiplication, needed since every pair of parties had a separate instance of an additively homomorphic encryption scheme. This is avoided in SPDZ by using a single instance of an SHE scheme, along with the distributed key generation and decryption procedures. The main bottleneck in SPDZ is therefore the zero-knowledge proofs of *plaintext knowledge* (which are also needed in BDOZ, but are less expensive). If the security requirements are relaxed to *covert security*, then the proofs can be replaced with inexpensive cut-and-choose methods [50]. Some recent works [13, 46] with new zero-knowledge proofs techniques may help to reduce the cost of active security, but the practicality of these approaches is not yet clear.

3.2.4 Triple Generation with Oblivious Transfer

An alternative approach to homomorphic encryption is to use oblivious transfer, as in the MASCOT protocol. This is simpler and much more efficient computationally, but has quite high communication costs, so is preferable to SPDZ when using a high bandwidth network such as over a LAN.

Like BDOZ, the parties first perform pairwise secret-shared multiplications, and then use these to obtain a triple via equation (1). A single 1-out-of-2 OT on strings can be used to create secret shares of the product of a bit and a finite field element. If elements of the field can be represented by k-bit strings then k string-OTs can be used to multiply two field elements. This method is done between every pair of parties to create the triple, and then again on each component of the triple to create the MACs. Active security can be achieved with a small constant factor overhead, in two steps: first, privacy amplification is used to remove any potential leakage caused by selective failure attacks on the OTs; secondly, a sacrifice check is performed where one triple is used to verify correctness of another.

This approach can be implemented very efficiently with OT extensions based on symmetric primitives using AES hardware instructions in modern CPUs, and in a LAN setting improves upon the performance of SPDZ using SHE by over 200 times. Because of the relatively large number of OTs needed, however, the communication cost is quite high so it may be less suitable over a very slow network.

3.2.5 Constant-Round Protocols

The multi-party protocols described so far all require a number of rounds of communication that scales linearly with the depth of the circuit, due to their reliance on secret sharing. Obtaining a *constant* number of rounds requires different techniques, and the most practical protocols that achieve this are all based on extending Yao's garbled circuits technique to the multi-party setting.

The basic idea is that the parties will first use a *non-constant-round* multi-party protocol (such as SPDZ) to produce a single, Yao-style garbled circuit, such that no single party knows all the wire keys of the garbled circuit. This can be done by executing the garbling algorithm in MPC. To obtain an efficient and constant-round protocol, we must be careful to ensure that the MPC protocol evaluating the garbling algorithm does not need to perform any PRF or encryption steps in MPC, as this would be prohibitively expensive. To do this, standard Yao garbling is modified to use n sets of wire keys, instead of just one, so that each party knows one set of keys per wire. In each garbled gate, the correct n keys for the output wire are encrypted using one key from each party for each of the two input wires, based on the encryption scheme (for gate g):

$$\mathsf{Enc}^{g}_{k_1,\ldots,k_n}(x) = \bigoplus_{i=1}^{n} F_{k_i}(g,i) \oplus x$$

where F is a PRF. Here, each party P_i holds a subkey k_i, and x denotes the set of output wire keys being encrypted. Notice that P_i can locally compute the PRF term in the above, so the only computation required of the MPC protocol is to obtain a secret-sharing of the correct output wire key, and XOR operations.

TinyOT plus BMR. In this approach [96], a multi-party version of the TinyOT protocol is used to multiply the secret-shared wire masks in each AND gate. When using the free-XOR optimization, the TinyOT MACs can then be used to directly obtain shares of the correct output wire key for each gate, because the MAC relation turns out to be exactly the same as the relation needed between keys and wire masks in free-XOR. After this, each party can compute an additive share of the entire garbled circuit, and then open this (and the appropriate input wire keys) to evaluate the circuit.

Note that the secret-shared garbled circuit is *unauthenticated*, and a corrupt party could tamper with it during the opening. However, any tampering will be detected during the circuit evaluation phase because each party checks for the presence of their own subkey when decrypting the output wire in each AND gate, which means it is not possible for the adversary to cause an incorrect output. Overall, the main cost of garbling a circuit is just one TinyOT multiplication per AND gate, i.e. this incurs no more communication than evaluating the same circuit using TinyOT (which would require a non-constant round complexity).

TinyOT plus authenticated garbling. Authenticated garbling [173] starts with a similar approach to BMR, where a general MPC protocol is used to construct the garbled circuit. TinyOT is also used to multiply the wire masks of AND gates, and [173] also introduced an optimization to TinyOT that

reduces the amount of cut-and-choose needed to generate multiplication triples. The main difference, compared with BMR, is that instead of using the TinyOT MACs to construct shares of a garbled circuit evaluated by all parties, they are used to create an authenticated garbled circuit that is opened to just one evaluation party, with the MACs used to ensure that this is done correctly.

The cost of authenticated garbling is roughly the same as in the BMR approach, since in both methods the dominant cost is creating the TinyOT triples. [173] presented implementation results for up to 128 parties running on Amazon servers, which could securely compute the AES circuit in a few minutes, whilst [96] described a BMR implementation for up to 9 parties with comparable results.

3.3 Verifiable Computation

In this subsection, we discuss work on the relation between verifiable computation and MPC. Verifiable computation aims to address the problem of ensuring correctness of an outsourced computation performed by an untrusted worker. In the most basic scenario, a client sends its input to the worker, the worker computes a function on this output, and sends the output to the client along with a cryptographic proof that the computation was performed correctly. In a slight extension to this scenario, the server is also allowed to insert information into the proof in "zero knowledge", i.e., the client does not learn anything about the information except that it was correctly used in the computation.

For this scenario to make sense, the verification that the client has to perform on this proof has to be easier than the computation itself; otherwise there is little point in outsourcing. There have been several results (see, e.g., [93, 69, 81]) that show it is indeed possible to give a zero-knowledge proof of any statement in **NP** (i.e., the server's computation can be as resource-intensive as solving an **NP**-complete instance) that has constant proof size and can be verified in a short time (sublinear in the size of the circuit that computes the computation). However, up to this point, the constructions were still believed to be impractical.

Verifiable computation was made practical due to a breakthrough result from 2013 [146] that used the result of [81] to provide proofs, so-called SNARKs (short for *Succinct Non-interactive ARguments of Knowledge*), that had proofs of fixed size 288 bytes (i.e., independent of the complexity of the computation), were verifiable in very short time (i.e., with verification effort sublinear to the computation's circuit size) and still reasonably efficient to build. Since this breakthrough result, the most well-known application of SNARKs has been in the blockchain setting: in the Zerocash system [18], a party performing a payment can efficiently prove to anybody that the payment was correct without revealing its details.

Approaches in the area of verifiable computation typically depend on key material that is dependent on the particular computation at hand: an evaluation key that the worker uses to build its proof, and a verification key that the client uses to verify its correctness. Both these keys have to be generated by a party that the client trusts: in practice, by a third party trusted by all clients in the system. To address this trust setting for Zerocash, MPC has been used in practice to generate these parameters in a distributed way without any party involved learning the so-called "trapdoor" that allows the generation of fake proofs [19].

A recent work [158] first discussed a privacy-preserving variant of the above setting. Instead of having one worker who gets the input and performs the computation on it, in this work the role of the worker is distributed between multiple parties in a privacy-preserving way using multi-party computation. Specifically, MPC is used not only to compute the function result (as in a normal application of MPC in the outsourcing setting), but also to build the cryptographic proof of correctness of the computation. In [158], both tasks are performed with semi-honest MPC in the honest-majority setting, e.g., the one-out-of-three setting of Section 3.1; as it turns out, building the cryptographic

proof in this setting can be done very efficiently. The security model one ends up with is that the privacy of the sensitive input data is only protected against attacks by a passive minority of the workers (due to the MPC protocols used), but correctness of the computation result is guaranteed even if all workers are actively corrupted. This work can also be extended to the setting where multiple parties provide input to the computation or receive its output.

With extensions of the above works, it is also possible to deal with the useful scenario where the verifiable computation not just uses input by the client(s), but also by third parties. This is relevant for scenarios such as smart metering [10], where a smart meter produces electricity usage data that a customer can then aggregate prior to sending it to the utility company: the customer wants to provide an aggregate to hide its detailed usage data but the utility company does not trust the client to compute this aggregate correctly. In this setting, one main desideratum is that the same input data can be re-used in multiple computations of different kinds; several solutions to this problem exist, [10, 71, 169]; the last one (an early result of the SODA project) also applies in case the computation is to be performed with MPC [169].

In practice, building a verifiable computation proof for a particular computation requires much more effort than performing the computation using semi-honest MPC, while it is competitive with actively secure MPC [158]. To reduce the overhead imposed by adding a verifiable computation proof to a MPC, the approach of verifiability by certificate validation has been proposed [57]. This approach avoids building a proof for the full computation by running a small "certificate validation" routine that checks that the result of a performed computation was correct, and only produces a proof that the result correctly passed this check. Whether this is possible depends on the availability of a easy-to-verify "certificate" of correctness of a computation; such certificates exist for instance for linear programming [57] and linear regression [79].

4 Data-Oblivious MPC

4.1 Oblivious RAM

The random access machine model (RAM) is a simple model of computation that closely captures the inner workings of modern computer architectures and many popular algorithms and real world programs designed for them. From a high level, a RAM program is a sequence of instructions coupled with access to an array of registers, which can store data. RAM programs are executed by executing one instruction after the other and reading or writing to the registers in between these instructions. That is, each instruction either outputs a location that should be read and taken as input to the next instruction or the instruction outputs a location and a value that should be written to it.

The main advantage of representing programs in the RAM model, in comparison to a circuits representation, is that the size of RAM programs does not depend on the size of the input data. A simple illustrating example is binary search, where the size of a circuit linearly depends on the size of the input data. In comparison, the size of a RAM program only has a logarithmic dependency.

Garbled RAM Programs. Secure computation of RAM programs has received a large amount of attention in the cryptographic research community. The main challenge in efficiently realizing secure computation of RAM programs is that the access sequence of a RAM program may depend on the contents of the registers. Hence, it is not to sufficient to just encrypt the RAM program and the registers, since the access sequence by itself can leak sensitive information. In [87, 88], the notion of Oblivious RAM (ORAM) was originally introduced. ORAM is a data structure, which represents an encrypted array. Read and write accesses can be performed obliviously, in the sense that an access to the ORAM data structure does not reveal anything about which element in the underlying array was accessed. From an adversarial point of view, this means that an observer that sees accesses made to the encrypted ORAM data structure, learns nothing about which elements in the encrypted array are actually accessed.

In [126], it is shown how to garble RAM programs. Their solution is based on ORAM, one-way functions, and a circular security assumption on Yao's garbled circuits. In [83], this result is further improved by providing two new constructions of garbled RAM. The first construction attains the same same asymptotic efficiency as the original construction from [126] and avoids the circular security assumption through the use of identity-based encryption. The second construction achieves a slightly worse asymptotic performance, but only requires the existence of pseudorandom functions and therefore one-way functions. In [77], the first black-box construction of garbled RAM is presented. Their result only requires the existence of one-way functions.

While the size of garbled RAM programs from standard assumptions is linear in the runtime of the RAM program, it has been shown in [38, 37] that using indistinguishability obfuscation, one can obtain garbled RAM programs that are only larger by a polylogarithmic factor in comparison to the plain program.

Unfortunately, while garbled RAMs could be very useful in practical MPC applications, all existing constructions (to the best of our knowledge) are only of theoretical interest and are prohibitively inefficient for practical applications.

Parallel Oblivious RAM. All of the above constructions are inherently sequential in the sense that all ORAM accesses need to be performed in a sequential fashion and cannot be parallelized. The main issue is that standard ORAM does not provide any obliviousness guarantees if accesses are performed in parallel. This issue was first addressed in [31], where the authors formalized the notion

of oblivious parallel RAM (OPRAM) and provide a first construction, which has a polylogarithmic slowdown in comparison to the best known ORAM constructions. This result has been improved upon in [40], where the authors provide an improved construction of OPRAM that is asymptotically as efficient as the state-of-the-art in ORAM in an amortized sense. In addition, the authors provide a generic construction of OPRAM from ORAM, which only loses a logarithmic factor in efficiency in an amortized sense. One disadvantage of [40], in comparison to [31] is that their OPRAM construction always requires every CPU to participate. More precisely, assume at some point during the execution of a parallel RAM program only a subset of all available CPUs would need to access the RAM. In [31] only the active CPUs need to participate in the OPRAM access. In [40], all CPUs always need to participate.

Zero-Knowledge Proofs for Garbled RAM Programs. Beyond research that has look at garbling general purpose RAM programs, there has also been some interest in garbling more specific RAM programs. One popular setting that has received some attention is that of zero-knowledge proofs and arguments over outsourced databases [97]. In this setting, a data owner outsources his data inside of an ORAM data structure to a server. Later on, the data owner proves statements that can be expressed as RAM programs over the outsourced database to the server in zero knowledge. It has been shown that this type of gabled RAM programs can be executed in a constant number of rounds in a straightforward manner. The main insight is that only the prover has input to the computation and that the prover can predict which ORAM accesses will be performed. Given these two facts, roughly speaking, it is possible to garble one circuit that contains all instructions and provide all ORAM accesses separately at the same time. The ORAM accesses and whatever they retrieve are part of the input to the garbled circuit. The circuit, internally, checks two things. First, it checks that the ORAM accesses that have been performed are indeed consistent, with the accesses the RAM program is supposed to do. Second, it checks that given the ORAM accesses, the statement expressed by the RAM program is valid. On an intuitive level, the server does not learn anything beyond the validity of the statement due to the privacy of the garbling scheme. The data owner is not able to cheat, since the circuit verifies the correctness of every ORAM access that was performed.

Efficiency Measures for ORAM. Due to its importance as a building block in various applications, significant research has been invested into making concrete ORAM instantiations as efficient as possible. Which performance metric is most important strongly depends on the concrete application in which ORAM is used. It is an active area of research to find ORAM constructions which are asymptotically as well as practically efficient. From a high-level perspective there are two applications for which ORAM constructions are being optimized.

The first one is secure computation, where the most important property for an ORAM construction is its "MPC-friendliness". Here, ORAM constructions are usually not just assessed by their asymptotic guarantees, but rather in connection to a concrete secure computation instantiation. The efficiency of secure computation of RAM programs crucially depends on how efficiently the read and write operations to the ORAM data structure can be realized in secure computation.

The second application is cloud storage. In this application, the most important performance metrics are storage and bandwidth overhead. The bandwidth overhead metric is concerned with how much data needs to be read from the ORAM data structure to read a single data entry in the underlying array. Most existing solutions follow the design principle of augmenting an access to a desired element with accesses to dummy data elements. The goal of these dummy accesses is to ensure the obliviousness of the current and of future accesses. For this type of approaches, the goal is to minimize

Construction	Storage Overhead	Online Overhead	Bandwidth Overhead
(deamortized) Square-root [89]	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(\sqrt{N}\log^2 N)$
(deamortized) Hierarchical [89]	$\mathcal{O}(N\log N)$	–	$\mathcal{O}(\log^3 N)$
Tree-ORAM [161]	$\mathcal{O}(N\log N)$	–	$\mathcal{O}(\log^3 N)$
Path-ORAM [164]	$\mathcal{O}(N)$	–	$\mathcal{O}(\log^2 N)$
Obliv. Storage [30]	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\sqrt{N}\log N)$

Table 6: Comparison of most influential ORAM constructions

the number of dummy accesses, and hence the bandwidth overhead of the ORAM data structure. The storage overhead performance metric is concerned with the size of the ORAM data structure in comparison to the size of the plain encrypted array that it represents. A comparison of the most influential results can be found in Table 6.

Constructions of ORAM. In [87, 88], Goldreich and Ostrovsky introduced the notion of ORAM, proposed the first two constructions thereof, and proved that a large class of ORAM construction is subject to a lower bound of $\Omega(\log N)$ on the number of actual accesses that are needed to simulate one oblivious access. This bound has been further examined in [32]. Both of their proposed constructions have sublinear amortized bandwidth overhead, but suffer from superlinear worst-case bandwidth overheads. Their first construction, the so-called square-root ORAM construction, is asymptotically not optimal w.r.t. its bandwidth overhead, but is conceptually very simple and has a small storage overhead. It has recently been shown that, due to its simplicity and despite being asymptotically inferior to other existing solutions, square-root ORAM can be very fast in practice for smaller to medium sized databases [180]. The second construction, the so-called hierarchical ORAM, is asymptotically more efficient w.r.t. bandwidth overhead, but suffers from a larger storage overhead, larger practical constants, and is conceptually more complex.

In subsequent work [141, 89], it was shown how to deamortize both constructions, i.e., how to make their amortized overhead be their worst-case overhead, at the cost of roughly doubling server's storage overhead. The deamortization procedure is based on the observation that, from time to time, both constructions need to be reshuffled completely. This complete reshuffling is the reason for the superlinear worst-case bandwidth overhead. Using this observation the authors suggest to keep two instances of an ORAM instead of one. The active ORAM is used for responding to accesses, while the passive ORAM is being reshuffled bit by bit at every access to the active one. Once the active ORAM needs to be fully reshuffled, the passive ORAM is just done with a full reshuffle. The active and the passive ORAM now switch roles and the previously passive, now fully reshuffled ORAM is used for responding to accesses, while the previously active one will be reshuffled in the background.

In a seminal result, Shi et al. [161] introduced a fundamentally new approach for constructing ORAM based on binary trees, which has a worst-case overhead of $\mathcal{O}(\log^3 N)$ and is computationally as well as conceptually simpler than previous works. In Path-ORAM [164], the tree-based approach was refined and the bandwidth overhead has been reduced to $\mathcal{O}(\log^2 N)$. As of today, this tree-based approach serves as the basis for many of the state-of-the-art solutions across many ORAM related research areas. The main idea behind the tree-based ORAM approach is to distributed all data elements in a binary tree data structure, where each node is an encrypted bucket that can hold several data elements. As an invariant, every element is assigned a leaf node and at every point in time the data element is somewhere between the root bucket and the destination leaf. Upon each access to

a data element, the corresponding path from root to leaf node is retrieved. The desired element is found, moved from its current location to the root node bucket, and assigned a new destination leaf node. From an intuitive point of view, obliviousness is guaranteed because upon every new access to the same element, a new uniformly random path is retrieved. To ensure that the root bucket does not overflow, tree-based approaches implement different flavours of an eviction procedure that pushes elements downwards towards the leafs as far as possible along their corresponding path.

Due to their simplicity and efficiency, [161, 164] have been the basis for many theoretical and practical follow-up works [43, 132, 153, 78, 60, 163].

ORAM in the Cloud Storage Setting. With the rise of cloud computing and a constantly increasing popularity of outsourced storage providers, like Dropbox, ORAM has also been used to provide better security and privacy guarantees against curious outsourced storage providers. As in the case of RAM computations, the main motivation for using ORAM was that the access pattern can reveal non-trivial information about the stored data, even if the data itself is encrypted. For instance, Islam et. al. [100] showed that by observing encrypted search queries over an encrypted database, an honest-but-curious server storing the database, could learn significant amounts of information about the search queries by just analyzing their access patterns.

Even though the security requirements for RAM computations and outsourced storage may seem similar in spirit, there are some noticeable differences. The most important difference is, that in the RAM computation setting, the RAM itself cannot perform any computations. In the cloud storage setting, the server, however, is able to perform computations. This difference has been used in different ways to overcome the Goldreich-Ostrovsky lower bound on the communication overhead, which only holds for ORAM that is not allowed to perform any computations on its own.

Path-PIR [129] uses server-side computations to achieve a practically very small, yet still poly-logarithmic bandwidth overhead. Their main idea was to combine Path-ORAM and techniques from the area of private information retrieval (PIR). In the PIR setting, roughly speaking, the client's goal is to retrieve a data element from a server, without revealing which. In particular, the client only wants to perform read (and no write) operations and the server is allowed to perform computations on the outsourced data.

In [5], Apon et al. formally define the notion of Verifiable Oblivious Storage, which generalizes the notion of ORAM by allowing the server to perform computations, and show that the ORAM lower bound does not apply to their setting by providing a scheme with constant overhead per access based on fully homomorphic encryption. Even though their scheme breaks the lower bound and only has constant overhead, their solution is, due to the use of fully homomorphic encryption, mostly of theoretical interest. In [60] a scheme, called Onion ORAM, is presented that breaks the lower bound, but only relies on additively homomorphic encryption. Their main technical tool for achieving such a small overhead is a bandwidth efficient homomorphic select operator that allows a client to select a single encrypted block from a list of blocks. Such a homomorphic select operator can be built generically from any additively homomorphic encryption scheme that fulfils some additional, yet mild, properties.

Another approach to circumvent the ORAM lower bound and to achieve practical efficiency in the outsourced storage setting, was introduced in [30] and then improved upon in [106, 153]. Their main idea was to split the total bandwidth overhead into two parts. The first part, the so called online overhead, is the amount of data that needs to be transmitted between the client and the server to retrieve a desired data element obliviously. The second part, the offline overhead, is the amount of data that needs to be transmitted between the two parties to ensure obliviousness of future accesses.

One can think of the offline overhead as background work that, usually, moves around encrypted data elements in the ORAM data structure to ensure the desired obliviousness guarantees. Splitting the total bandwidth overhead this way and then minimizing the online overhead has practical advantages. It allows the client to efficiently retrieve data from the server without much latency during bursts of requests and then do the background work during quieter phases.

In [30], Boneh et al. presented a solution, for a primitive strongly related to ORAM, that has $\mathcal{O}(\log N)$ online and $\mathcal{O}(\sqrt{N}\log N)$ worst-case overhead. Their construction can be considered a hybrid between the original square-root ORAM and hierarchical ORAM solutions. This idea has been refined and improved in [106], where the authors provide Burst ORAM, a practically significantly more efficient construction based on the hierarchical ORAM construction. In particular, their construction has a constant online and a logarithmic offline bandwidth overhead. In addition, this work introduces the so-called XOR trick, which allows a client to retrieve a single real block, which is hidden in a set of dummy blocks without any overhead. The basic idea is that the server transmits the XOR of the dummy values and the real block. The client can recompute the dummy blocks locally using a pseudorandom function and then obtain the real block by XORing out the dummy blocks from the received value. This approach has been further refined and combined with the popular tree-based Path ORAM construction in [153]. Their resulting construction, Ring ORAM, has a computational load on the server that is comparable to that of Burst ORAM, but achieves a practically smaller bandwidth overhead.

A recently proposed ORAM construction called Circuit-ORAM is based on Path-ORAM and highly optimized for secure computation settings. Their main idea is a new ORAM construction that is not optimized w.r.t. the bandwidth overhead, but rather w.r.t. the circuit size. As such it is highly suitable for being used inside of garbled circuits. Circuit-ORAM has been implemented and is being used as part of a secure computation framework called ObliVM [124].

4.2 Oblivious Data Structures and Basic Operations

To effectively build MPC-based applications, it is important to have efficient MPC-based implementations of common *abstract data types* (ADT), which we call "oblivious data structures".

Above, we gave a survey of oblivious RAM and its application to MPC. As we have seen, using ORAM in MPC can be seen as providing an oblivious implementation of an array. More abstractly, it is an implementation of the "vector" abstract data type (of which an array is the standard implementation) that captures a set of elements that can be accessed by integer indices.

In this subsection, we discuss oblivious data structures implementing several other common abstract data types such as maps, stacks, queues, and graphs (see, e.g., [90] for a general, non-MPC discussion of common abstract data types). It is not trivial to build such oblivious data structures, as the ORAM case shows: whereas on a normal processor, accessing an array at a certain index takes constant time, with MPC even this simple operation requires at least logarithmic work if the index needs to be hidden. As in the traditional setting, different oblivious data structures implementing the same abstract data type can be useful depending on how many items are stored, and what operations are executed how many times. Apart from this, MPC-based implementations can also characterised according to several other criteria:

- *What information does the data type leak?* Although an MPC-based implementation of an abstract data type aims to hide at least some of the information it contains, it does not necessarily hide everything. For instance, a MPC-based implementation of a graph may have a public set of vertices and a private set of edges, or public vertices and edges and private weights. Note

that, at the very least, an MPC-based implementation of an abstract data type has to leak some upper bounds on its size, e.g., the number of items in an array.

- *How is the data partitioned?* Normally, MPC-based abstract data types assume that all sensitive data they contain are kept hidden from the participants in the MPC protocols. We call this the "distributed" setting. However, variants are possible where the data originally came from the protocol participants and is available to them in the plain, which typically allows operations to be executed faster. For example, matrices can be horizontally partitioned, meaning that each protocol participant holds a number of rows of the matrix, or vertically partitioned, meaning that each protocol participant holds a number of columns.

- *With which MPC approaches is the data type compatible?* MPC-based implementations of abstract data types may be specific to particular approaches for MPC discussed in the previous sections. One important distinction is between implementations based on binary circuits (that compute on bits) or arithmetic circuits (that typically compute on numbers modulo a large prime), that have different performance characteristics. Another important distinction is between semi-honest and active security. Assuming that parties stick to the protocol descriptions usually allows for significantly faster implementations, as it is not necessary to detect all possible kinds of misbehaviour. Finally, implementations may use auxiliary cryptographic primitives that are specific to a particular setting, e.g., a particular number of parties. We will refer to approaches that work regardless of the above differences as "generic".

Below, we survey the literature on MPC-based abstract data types with these criteria in mind. When it is not explicitly mentioned, the implementations below are generic, in the distributed setting, and only leak an upper bound on the number of items in the data structure.

A final important criterion is concurrency. Especially in the big data setting, performance of MPC can be improved by distributing the work of a single protocol participant between multiple processors or machines. Hence, the question is to what extent different operations on the same abstract data type can be carried out in parallel, and possibly, if the storage of the data can be distributed between multiple machines. Unfortunately, application-level concurrency is a rather new topic for MPC that is typically not considered in the design of MPC implementations of abstract data types. For this reason, we survey works in this direction separately in Section 4.2.7.

We remark that there is also research on oblivious data structures in settings other than MPC. For instance, as discussed above, oblivious RAMs were not originally designed to be a MPC-based implementation of the "vector" abstract data type, but instead to hide access patterns to external storage. We briefly discuss these other kinds of oblivious data structures and their relation to MPC in Section 4.2.8.

4.2.1 Vector

One of the most simple ADTs is the vector, a linear sequence of n elements that are accessed by integral indices 0 to $n-1$. This corresponds directly to the array data structure. Vectors accessed by public indices are trivial to implement in MPC, so we focus here on the setting where the vector needs to be indexed with a secret index; usually the values are also secret (although for read-only vectors, it can also make sense to consider public values). (Note that, in MPC, all elements in an array have to have the same size, otherwise the size of the retrieved item leaks information about which index was accessed.)

The folklore MPC implementation of the vector ADT is the so-called "linear scan". That is, to retrieve from array $(a_1, \ldots, a_n)$ the item at index i, the index is represented as a bit vector $b_1, \ldots, b_n$

with a one at the ith location and the element is retrieved by computing the inner product $b_1 \cdot a_1 + \ldots + b_n \cdot a_n$. In this setting, the size of the array is the only thing that is leaked; data is fully distributed; and it is applicable to any type of MPC. This is the standard implementation that other solutions are benchmarked against; for instance, in the arithmetic actively secure setting [110] or the binary passively secure setting [180]. Note that indices need to be represented as bit vectors for the array access operation, but this is an inefficient representation to manipulate or store indices: hence, array indices can be represented as numbers and converted to bit vectors in the event of an access (e.g., [54]; in some settings, representing indices as roots of unity is more efficient [56].

More generally, an oblivious RAM can be seen as a direct implementation of the vector ADT; and indeed, the linear scan is sometimes referred to as the "trivial ORAM" [110]. ORAMs typically only leak an upper bound on the number of items in the dictionary, and can be compatible with any type of MPC approach, for instance ranging from passively secure 2PC based on binary circuits [184] to actively secure multi-party computation based on arithmetic circuits [110]. We refer to Section 4.1 for details, mentioning here only a recent development important when the Vector ADT is used in a particular setting: namely, [64] shows how to optimize a MPC-based ORAM for the operation `a[i]=`f`(a[i])` (i.e., retrieving and updating an array element) in case f is relatively complex.

In the particular setting of passively secure 2-party computation where both parties hold a part of the vector in the clear, a particular class of algorithms operating on vectors can be run more efficiently [160] than with the above implementations. The idea is easily explained for computing the median of a vector of numbers [3]: the two parties compute the median of their parts, compare these medians with MPC and open the result of the comparison. The party with the lower median removes the lower half of its dataset and the party with the higher median removes the higher half of its dataset, and the parties proceed recursively. This approach can be applied to problems that are so-called "greedy-compatible" [160], including the computation of a minimal spanning tree on a list of points on a plane.

Another particular setting where a more efficient solution exists is where vector reads are at secret locations, but writes are at public locations. Under these conditions and assuming a passive adversary (and multiple parties), the approach from [72] implements the vector ADT by making use of a custom-designed cryptographic primitive: "multi-party oblivious transfer".

4.2.2 Dictionary/Map/Associative Array

A dictionary, also known in the literature as an associative array or map, is a collection of elements that can be accessed by means of a key. Typical non-MPC data structures implementing this ADT are hash tables or different kinds of search trees. As with vectors, implementing a dictionary with MPC becomes interesting when the key used to access elements is secret. Recall from above that ORAM is the data-oblivious version of the vector ADT, which has indices $0, \ldots, n-1$ whose items are always defined. If keys are integers, then the dictionary ADT is almost the same except that the item at a given index is not always defined. Because of this analogy, the data-oblivious variant of a dictionary is sometimes referred to as a "non-contiguous ORAM" [110].

The trivial way of implementing a dictionary (compatible with any type of MPC, hiding only an upper bound on the number of items, and in the distributed data setting) is with a "linear scan" technique [110]: keys and values are stored together in an array, and to access the element with a given key, we traverse the array and compare the stored key with the key to be retrieved. In [110], it is also shown how to avoid comparing the key to be retrieved k with all stored keys k_i, basically by performing binary search in a multiplication tree with $k - k_i$ as leaf nodes; this is still linear in the maximum size of the dictionary but it reduces the number of comparisons from linear to logarithmic.

In [181], another implementation of the dictionary ADT is given that also stores the (key,value) pairs in an array, but keeps them sorted. They key observation is that multiple independent update operations can be batched together by means of two sorting steps: one sorting the keys to group together all update operations operating on the same key and hence find the latest version for the value of each key, and one to move these latest values to the beginning of the list and "dead" values to the end. (While implemented in the semi-honest binary 2PC setting of garbled circuits, the techniques are generic and should apply to any MPC setting.)

4.2.3 Set/Multiset

A set is a collection of distinct elements, without any particular order, on which the typical operation is to check occurence of a certain given element. In a multiset, each element also has a multiplicity that intuitively counts how often it occurs. Typical non-MPC implementations are with hash tables or sorted arrays (with binary search to test for membership).

A large body of works addresses the problem of "private set intersection" (PSI). In this problem, two parties each have their own private set, and they want to both learn their intersection. Many solutions exist, both in the semi-honest and the malicious settings, and both based on custom protocols and implemented as circuits in MPC [98]. The state-of-the-art of PSI is discussed in [149], where also extensions of this problem are discussed where the outcome of the algorithm does not need to be the set intersection itself, but can also be the application of any symmetric function on the intersection. Many implementations of PSI exist, for instance in Obliv-C [182]. In the same distributed setting, other set operations such as the union have also been considered, e.g., see [33]. For more information, we refer to Deliverable D2.1 where private set intersection is discussed at length.

The general set ADT can be implemented efficiently in MPC based on Bloom filters, if a certain amount of false positives for set membership can be allowed. In a Bloom filter, occurence of items is stored in bit array a of length m. Given hash functions $h_1, \ldots, h_k$ with range $\{1, 2, \ldots, m\}$, an element e is added to the Bloom filter by setting $a[h_i(e)]$ to 1 for all i. Choosing m and i approriately allows to control the false positive rate. Bloom filters are generic but concrete implementations exist, e.g., in the Sharemind framework [26].

Generically, the set ADT can be implemented on top of the vector ADT by storing the values sorted in a vector, and performing binary search to test for membership. This implementation does not suffer from the false positive problem sketched above. In this context, the work of [82] is particularly relevant since it shows how to efficiently perform binary search based on a sorted ORAM, thus effectively giving an implementation of sets. This is a generic approach that can be combined with any type of MPC. Another implementation of the set ADT in the setting where one party holds the array and the other party holds the item to look for, would be based on the work of [123] that considers binary search exactly in this setting.

4.2.4 Queue, Stack, and Deque

Queues and stacks are ADTs that represent a sequence of items that can be manipulated at the beginning or the end. In a queue, elements are added at the end and removed at the beginning (known as FIFO: fist-in, first-out). In a stack, elements are added and removed at the beginning (known as LIFO: last-in, first-out). Typical non-MPC implementations of these ADTs are based on singly-linked or doubly-linked lists. Clearly, if the access pattern to these ADTs is public, then MPC implementations of these ADTs are trivial: one can simply store the encrypted or secret-shared data inside a non-MPC implementation of the respective ADT. In MPC, however, it is often needed to be able to

perform "conditional operatations", e.g., a push operation with a secret flag to indicate whether the operation should actually be performed or not. In particular, in this case, the number of items currently in the data structure remains hidden. We now discuss MPC-based implementations of these ADTs supporting such conditional operations.

In [181], MPC-based implementations of the stack and queue ADTs are given, both based on similar ideas. The basic idea of the stack is to have multiple "levels" of items, each time double the size of the previous. Roughly, a single conditional push is performed by adding the item to the first level, but after two conditional pushes, if necessary (namely, if both pushes were real) the items are obliviously both moved to the next level. Similarly, once every four conditional pushes, if needed, items are moved from level two to level three. Pops are implemented by similarly moving items back from higher to lower levels in a timely fashion. The queue ADT is implemented by having two stacks, one for adding items and one for removing items, of which the highest levels are connected to each other.

Stacks and queues can also be implemented based on ORAM. The implementation of the Gale-Shapley algorithm described in [110] includes a straightforward ORAM-based implementation of a stack with a conditional add operation. ObliVM [124] also provides MPC implementations of stacks and queues, that combine ORAMs with ideas from [174]. The stack keeps track of the index in the ORAM of the top of the stack, while each element in the stack is stored along with the index of the next item. It turns out that this leads to an access pattern to the ORAM (a "rooted trees with bounded degree") that can be efficiently implemented due to an optimization by [174]. The queue implementation is similar.

In [64], a data structure similar to stacks and queues is implemented called a "linked multi-list". This data structure allows traversing through multiple lists, while at each access hiding where the current element came from, i.e., implementing $x \leftarrow a[i, \mathrm{pos}[i]]; \mathrm{pos}[i]{+}{+}$ where i is secret.

4.2.5 Priority queue

In the priority queue ADT, elements are added with a certian priority, and there is an efficient operation to retrieve the element with the smallest priority value. The most common non-MPC implementation is a heap, but other implementations, for instance with an array, are also possible.

The first MPC-based implementation of a priority queue is the generic approach due to [168]. The construction is based on a "bucket heap": multiple levels of sorted lists with associated bufffers, both of whose sizes double going from one level to the next. Loosely, items are inserted into buffers, starting at the first level and obliviously moved to the next level if the previous one is full.

An alternative implementation, due to [110], is based on putting a heap inside an ORAM. Recall that a heap is a binary tree satisfying a certain "heap" property; this binary tree is simply stored inside an ORAM by storing the root at location 0, its two children at locations 1 and 2, and so on. The ObliVM priority queue implementation [124] stores a heap inside an ORAM in the same way, but, as its stack above, additionally includes the optimizations of [174].

4.2.6 Graphs

The problem of performing MPC algorithms on graphs has received wide attention in the literature. Discussing these algorithms is out of the scope of this section (that is on data structures and not algorithms), but for completeness we do survey the kind of representations of graphs that these algorithms are based on. Recall that in the non-MPC setting, graphs are typically represented by adjacency lists (per vertex, a list of adjacent vertices), adjacency matrices (each row and each column represents a

vertex and cells represent the adjacency relation between the respective vertices), or incidence matrices (where rows are vertices, columns are edges, and cells show their relation).

Adjacency matrices have been used in [21] to perform breadth-first search and computation of shortest paths, spanning trees, and maximal flow. Note that an adjaceny matrix representation implies that the set of vertices of the graph is public information. [180] combines an adjacency matrix with oblivious arrays and queues to perform breadth-first search more efficiently. Also ObliVM [124] uses adjacency matrices to perform depth-first search. In [4], also several graph algorithms are implemented on top of an adjacency matrix representation (in the semi-honest setting).

Several novel representations of graphs have been designed that do not directly have non-MPC counterparts. In [110], a graph is represented as a "sparse digraph": a single list of neighbouring edges of all vertices, with the edges for different vertices separated by a special "stop bit". Using this representation, the only thing leaked about the graph is the total number of edges and vertices. This representation is used, among other things, to compute shortest paths using Dijkstra's algorithm. In ObliVM [124], sparse graphs are represented by putting the concatenation of the adjacency lists in an ORAM, and this representation is used to compute shortest paths and minimal spanning trees. In this case, the total number of vertices is assumed to be public. In GraphSC [135], which will be discussed in more detail in Section 4.2.7, graphs are represented by a list containing both edges and vertices, with a flag indicating the type. This list is distributed between multiple paired computation nodes that perform computations on different parts of the list in parallel. This just leaks the total number of edges and vertices. Finally, [4] represents a graph as a list of weights for each edge; hence, the graph structure is assumed to be public except for these weights.

While the above representations share the full graph between the parties (i.e., they are in the distributed setting), there are also works in the non-fully-distributed case. In [123], the setting is considered where one party has a plaintext graph (represented by an adjacency matrix) and another has a plaintext source/destination pair, where the goal is to find the shortest path between them. Similarly, [33] considers the setting where two parties each have a plaintext graph, and the objective is to perform MPC (in the semi-honest model) on their joined graph. Finally, [75] considers the problem to build a public joined graph from private subgraphs held by the respective parties, without leaking which part of the resulting graph came from whom.

4.2.7 Oblivious Concurrent Data Structures

We now focus on MPC-based concurrent data structures. A concurrent data structure is a data structure designed to allow (some of) its operations to be executed more efficiently using parallelization. In MPC, this topic has not been extensively studied. Recall that in MPC, high-level algorithms are translated to a lower-level (binary or arithmetic) circuit representation for execution. At this lower circuit level, parallelizability has been considered [35], and parallelizability in a single node is already part of the inherent design of our FRESCO framework. However, low-level parallelizability only works well insofar as higher-level algorithms have been designed to be executable in parallel (or already have this property inherently) [135]. One work does consider parallelizability and MPC in a single framework, but unfortunately parellizability is only applied to compute the inputs to the MPC, which is still executed between a single set of nodes [170].

In fact, our literature study turned up only one work that specifically aims to design oblivious concurrent data structures, clearly indicating a gap in the state-of-the-art for SODA to address. This work, [135], considers the semi-honest 2PC setting based on binary circuits. The work focusses on graph algorithms on "data-augmented" directed graphs, i.e., graphs that have data associated with their vertices and edges. A framework is designed for efficiently parallelizing applications that follow

a "scatter-gather" paradigm where first the data at each edge is updated from its incoming vertex; and then the data at each vertex is updated from its incoming edges. As discussed above, a graph is represented as a list containing both edges and vertices, with a flag identifying the type. This list is distributed between arbitrarily many pairs of computation nodes. With a combination of circuit-level and algorithm-level parallelizatition, scatter-gather computations are implemented efficiently based on this representation, with applications including PageRank and matrix factorization.

4.2.8 Other types of oblivious data structures

For clarity, we now briefly contrast MPC-based oblivious data structures with two other data structures that have also been called *oblivious* in the literature. Perhaps the most relevent type are oblivious data structures in the "external-memory model" [174]. In this case, the objective isn't to hide sensitive information from the parties executing the algorithm, but to hide access patterns from an external memory accessed by the algorithm. In many cases, this means not just hiding which data is accessed, but also which operations are executed on this (which is something that MPC typically does not try to hide). On the other hand, oblivious algorithms in this model do allow the execution trace (apart from memory accesses) to depend on sensitive data (which MPC aims to prevent). Oblivious RAM is exactly the problem to implement a vector as a oblivious data structure in the external-memory model. Because of the similarity of the problems, oblivious data structures in this model can sometimes be turned into oblivious data structures in the MPC setting. As discussed, oblivious RAM is a prominent example, but also the external-memory oblivious data structures from [174] have been adapted to the MPC setting [124].

Another related notion is that of cache-oblivious algorithms, where the goal is to implement algorithms in such a way that the CPU cache is used efficiently, but without letting the algorithm depend on the size of the cache. This kind of algorithm does not appear to have a link to MPC.

5 Differential Privacy

If secure multi-party computation has to be used in practice, on actual sensitive data, it is paramount to understand what kind of outputs is *safe* to compute and release in a way that is not detrimental to privacy. In order to do so, we need a formal framework to reason about private information and the way it can leak. We need sound measures to quantitatively assess the privacy loss that users suffer by participating in distributed computations, and tools to prevent these losses to exceed tolerable bounds.

In this section we review existing work in this area focusing on *differential privacy (DP)* [67], which is a well-established notion of privacy that can help us in reasoning, in a mathematically sound way, about the information that leaks by running secure multi-party computation protocols over private data. This topic lies at the intersection of WP1 and WP3, and we have therefore decided to include it in the WP1 deliverable.

5.1 Introduction

Security of MPC. In previous sections we have discussed the state-of-the-art protocols for secure two- and multi-party computation. As described, these protocols allow a set of parties $P_1,\ldots,P_n$ with private inputs $x_1,\ldots,x_n$ to evaluate some function f (mutually agreed upon by the parties) on their secret inputs in such a way that the joint output z is indeed $z = f(x_1,\ldots,x_n)$ (even when some of the parties arbitrarily deviate from the protocol specification) and that any coalition of corrupt parties will not learn any *unnecessary* or *extra* information about the inputs of the honest parties. Note that here we put the emphasis on *unnecessary* or *extra* since it is not possible to ask that the corrupt parties should learn *nothing* about the inputs of the honest parties. Since the output z is computed from the inputs $x_1,\ldots,x_n$ it is natural to assume that z contains some information about the inputs. After all, if z did not depend at all on the input of some of the parties, why would these parties participate in the protocol in the first place?

Formally, the definition of secure multi-party computation guarantees that anything that can be learnt by a collusion of players $S \subset \{1,\ldots,n\}$ could have been *simulated* (e.g., efficiently computed in polynomial-time) by having access only to the output z and the set of the inputs of the corrupt parties $\{x_i | i \in S\}$. So, as the output necessarily contains some information about the inputs of the honest parties, and since the adversary can combine the output with their input it is natural to ask whether the privacy guaranteed by secure multi-party computation protocols is good enough in practice.

What vs. How. The discussion can be rephrased as follows: while secure multi-party computation is very useful in determining *how* to perform a certain distributed computation, secure multi-party computation does not say anything about *what* kind of distributed computations should be performed if one is interested in privacy.

As a simple example, consider the following secure two-party computation scenario: Alice and Bob want to compute the average of their salaries, but they care about the privacy of their inputs and therefore choose to use a state-of the art secure two-party computation protocol. They input their salaries x and y and they both receive the output $z = (x+y)/2$. The secure two-party computation protocol guarantees that they learn z *and nothing* else about the input of the other party. But is this good enough? Clearly not. Since Alice knows z and x, she can easily reconstruct the input of Bob as $y = 2z - x$. Therefore, even when using a secure two-party computation protocol, we have no guarantees that the privacy of the inputs of the honest parties is actually guaranteed unless we talk about what kind of function should be computed.

Privacy in isolation? Clearly, the previous example is very extreme. In practice, one could imagine that if the same statistics were computed over a large set of parties (say, thousands of employees) and if one assumes that the adversary only has a priori information about the salary of a subset of the parties (say, the adversary already knows the salary of half of the employees), then the adversary would not be able to completely recover the inputs of the other parties, but only learn some aggregate statistics about the inputs of the honest parties (in this case, the average salary of all the honest parties) which in many applications might be considered an acceptable situation. But what if the computation is run multiple times? What if the data entry of an employee is used in different secure multi-party computation protocols? How much information could be inferred by combining all the outputs from these computation?

It is often argued that as long as secure multi-party computation is used to compute functions which are not easily invertible, then some meaningful notion of privacy is preserved. As an example, consider *oblivious polynomial evaluation*, with Alice providing as input a non-trivial polynomial $p(\cdot)$ and Bob providing as input a point x and learning $y = p(x)$. Since a single point is not enough to reconstruct the polynomial, this could seem an acceptable situation. However, in this example it is easy to see that problems would arise if Alice was to reuse her input over multiple executions of the same protocol: if the adversary is allowed to learn $d+1$ points of a degree d polynomial $p(\cdot)$, then the adversary would completely learn the input of the honest party. These examples show that when considering privacy it is not enough to worry about what a single execution of a given protocol might leak about an input, but one has to be careful about how the information that leaks over multiple executions of (possibly different) protocols can be combined and used to compromise the privacy of the honest parties, taking also into account that the adversary might already have some a priori knowledge about the private inputs of the honest parties.

5.2 Early Definitions of Privacy and Anonymity

Before discussing differential privacy, we will start with a quick review of existing privacy tools which were proposed mostly within the database research community, and discuss their limitations.

The Model. In the remaining of the section we will consider a setting in which all parties $P_1, \ldots, P_n$ have already provided inputs to a secure multi-party computation protocol or, alternatively, to a trusted party T. As we know that secure multi-party computation protocols provide the same security guarantees as a trusted party, this will not make much of a difference. For consistency with the literature in this field, we will refer to the collection of all the private inputs $(x_1, \ldots, x_n)$ as a *database*, and sometimes refer to the individual inputs as *records* in the database.

The privacy and anonymity problem in databases can typically be described in the following way: a trusted collector T who owns a database D applies some transformation to the database D before releasing the database to the public. A database that has undergone such a transformation is usually referred to as a *sanitized* version of the database. The transformation should be such that, by looking at the sanitized database, it should not be possible to recover private information about individuals. At the same time, the sanitized version of the database should still enable to extract useful information about the aggregate population represented in the database (e.g., statistics).

Some (poor) attempts at sanitizing a database prior to release include:

Removal of Names, SSNs, etc. The simplest (and not very effective) way of sanitizing a database is to remove any information that can be directly used to identify individuals. This practice includes removal of trivially identifiable information such as names and social security numbers

from the database. These columns are then replaced with random identifiers to allow one to still be able to link records and entries corresponding to the same individual. The main issue with this approach is that, in some sense, *everything is an identifier*. This was first shown by Sweeney [166] in a celebrated result that showed the inadequacy of the technique by recovering (among other things) medical data of the governor of Massachusetts. In this study Sweeney analyzed a sanitized database which had been released by hospitals in Massachusetts. The database had been released to allow medical research. To preserve the privacy of the individuals in the database, all trivially identifiable information (such as name, full address and social security number) had been removed prior to release. However the database still contained postal codes and date of births, since this data is relevant to medical research (in order to be able to correlate medical conditions with area of living and age). What the hospital did not consider is that in the U.S. the registry of voters is publicly available and that such registry contains name, date of birth and postcode. Therefore, by joining the two databases it is possible to link individuals to their "anonymized" medical data. In the more blatant cases date of birth and postcode represented a unique identifier in the registry of voters (thus allowing the researcher to guess with certainty the link between the two databases). Even when this did not constitute a unique identifier it was still possible to link entries in the medical database with very small groups in the database of voters, thus still disclosing sensitive information.

The fact that removing trivially identifiable information is *not* an adequate measure to preserve privacy has been proven over and over. Some of the most famous examples include the Netflix incident [134] and the AOL incident [11]. In the Netflix incident, Netflix released information about users' ratings but removed usernames. By linking this database with public movie ratings that users willingly posted on other websites (e.g., IMDB), researchers were able to learn about users' ratings of movies that they had *not* reviewed yet on public platforms (while a user might be willing to rate "Lord of the Rings" publicly, they might not be interested in disclosing that they are interested in movies with more controversial content that might disclose their political or sexual orientation). In the AOL incident, AOL released an "anonymized" database of search queries. However, the search queries themselves contain enough information to allow one to completely deanonymize certain individuals (e.g., some search queries might disclose one's address, age, sex etc., which allows the researcher to uniquely identify an individual. This allows one to match individuals to search queries on possible sensitive topics such as medical conditions, sexuality, and political positions).

***k*-anonymity and other syntactical properties:** It has been argued that the previous attacks were possible only because it was possible to find relatively small groups (singleton in the worst case) to which certain individuals belong. As an answer to this problem the notion of k-anonymity was introduced [155]. In a nutshell, k-anonymity guarantees that, after the database has been sanitized, every set of *quasi identifiers* (e.g., every set of attributes that can be used to partially identify individuals) must appear at least k times in the sanitized database. To construct a k-anonymous version of a database one can *suppress* or *generalize* records i.e., one might choose to completely remove quasi identifiers (such as name and address) or to generalize some attributes to reduce the resolution of the data (e.g., remove the last significant digit of one's age). Since attacks are still possible against k-anonymity, several refinements have been proposed, including ℓ-diversity [127] and t-closeness [121].

We conclude that none of the above techniques is sufficient to achieve a desirable level of privacy. We do not report here on the long series of attacks that have been proposed on systems using the

above notions. Instead we conclude that the main problem behind these notions is that they only restrict the format of the *output* of the sanitization process, and they do not impose any requirement on the *process* itself. As an example, a trivial way to construct a k-anonymous database is to take any database D and then output a database D' where every record from D has been repeated k times. Now, it is clear that D' satisfies k-anonymity (since every set of *quasi identifiers* appears exactly k times), but it is also clear that releasing D' is equivalent to releasing the original database D.

5.3 Differential Privacy

To address the concerns described in the previous subsection, in the early 2000s the research community proposed a number of definitions and methods which finally resulted in the celebrated notion of DP [67][1]. In this subsection we present some basic definitions and tools for differential privacy. Several definitions are given in an informal style. We refer to the original literature for more details.

Model and Definition. In the DP setting we have a database D and a curator running a (randomized) mechanism M before revealing the output $M(D)$ to the public. The mechanism M computes (an approximation) of a target function $f(D)$.

The notion of DP considers an *adversary* A who observes the output of the mechanism $M(D)$ and tries to infer information about individuals in the database D. In this sense the notion of DP is *user centric* and and provides *worst-case* guarantees, since it requires privacy even against an adversary who already knows the entire database except for a single row. Therefore we introduce the following notion.

Definition 5.1 (Neighbouring Databases) *Two databases D, D' are said to be* neighbouring *if they differ at most in one record.*

There are two classical ways of concretely defining neighbouring databases:

1. The two databases must have the same number of records, and the value of a single record differs between the two databases;
2. We allow for erasures e.g., if D' is constructed by removing a record from D then we still consider D and D' to be neighbouring.

Both notions are considered in the literature, and the question of which definition is the "natural" one cannot be answered without considering the concrete application (and therefore, the desired privacy guarantees). Intuitively, the first notion guarantees privacy of the *data* of the individual (e.g., the adversary will not be able to determine if an individual voted yes or no) while the second notion hides even whether the individual has a record in the database (e.g., the adversary will not be able to determine if an individual voted yes or no). Anyway, it can be proven that any differentially private mechanism for one of the notions is also a differentially private mechanism for the other notion, with a change in the parameter of at most 2, therefore the actual notion will not make any difference in our setting.

Definition 5.2 (ε-Differential Privacy) *A mechanism M is said to be ε-differentially private if for all pairs of neighbouring databases D, D', for all unbounded adversaries A:*

$$\Pr[A(M(D)) = 1] \leq e^{\varepsilon} \cdot \Pr[A(M(D')) = 1]$$

[1] Several awards have been granted to the authors of the DP paper, including the prestigious Gödel Prize in 2017.

At first it might appear confusing that privacy parameter ε appears in the exponent, since then the factor e^{ε} grows exponentially in ε with the risk of making the bound quickly trivial (e.g., the right hand-side quickly becomes > 1). It has to be noted that differential privacy is considered for very small values of ε for which one can use the convenient approximation:

$$e^{\varepsilon} \approx 1 + \varepsilon$$

Note finally that the choice of the base e is completely arbitrary and in some papers the term e^{ε} is replaced e.g., with 2^{ε}. (The choice of e simplifies certain arguments since the derivative of e^x is e^x itself).

From an intuitive point of view, the definition of differential privacy says that the probability of any event cannot change by more than a factor ε as a function of any individual's private data. As an example, in a dictatorship, if D is a database of yes/no votes, and $M(D)$ is the result of a differentially private election, the probability that an individual will be incarcerated (the process is modelled as an algorithm A) because they voted against the dictator will be almost the same as the probability that the same individual will be incarcerated if they had voted in favour of the dictator.

Basic Composition. The advantage in using the e^{ε} term in the definition (instead of immediately plugging the more intuitive $1+\varepsilon$) is that, thanks to the properties of exponentiations, it is easier to reason about the composition of differentially private mechanisms. In other words, since $e^{\varepsilon_1} \cdot e^{\varepsilon_2} = e^{\varepsilon_1+\varepsilon_2}$, the following holds:

Lemma 5.3 ([66]) *A mechanism that permits n adaptive interactions with mechanisms $M_1, \ldots, M_n$ which are respectively $(\varepsilon_1, \ldots, \varepsilon_n)$-differentially private results in a $\sum \varepsilon_i$-differentially private mechanism.*

5.3.1 Other Definitions.

There are several variants of differential privacy in the literature. We mention some here:

Personalized Privacy [105]: In this version every record in the database has a different privacy value ε_i associated with it.

(ε, δ)-Differential Privacy: In this version of the definition it is allowed that the two probabilities in the definition differ of an additional additive factor δ i.e., the new condition states that:

$$\Pr[A(M(D)) = 1] \leq e^{\varepsilon} \cdot \Pr[A(M(D')) = 1] + \delta$$

An important difference between standard DP and this variant is that in the original definition if the left hand side of the equation is 0, then the right hand side must be 0 too, while using the revised definition a priori impossible events are assigned a (small but non-zero) probability (e.g., the probability of an insurance price rising is 0 if an individual does not participate in a medical study, but if the individual participates in a medical study it might happen with probability 10^{-6}).

Computational Differential Privacy [131]: In this variant we restrict our attention to computationally bounded adversaries A. In this case the additive value δ described above typically represent the probability that the cryptographic protocols employed are broken (e.g., the adversary correctly guesses the secret key).

5.4 Differentially Private Mechanisms

Several differentially private mechanisms have been proposed in the literature. In this subsection we review some of the most important ones. Note that the definition of differential privacy does not contain any requirement about the accuracy of the mechanism. Therefore the *constant mechanism* $M(D)$ which outputs some constant c for all databases D will trivially satisfy the definition of differential privacy but will produce completely useless results. Therefore the main question to be addressed when designing differentially private mechanisms for some task is not a *feasibility* question (i.e., are there any differentially private mechanisms for a function f?) but an *efficiency* question (i.e., can we construct a differentially private mechanism M that reasonably approximates the function f?).

5.4.1 Laplace Mechanism

The simplest differentially private mechanism is most likely the Laplace mechanism [67]. The mechanism is so called since it adds noise to the output according to the Laplace distribution, which is defined as:

Definition 5.4 (Laplace Distribution) *The distribution* $\mathrm{Lap}(\mu,\sigma)$ *parametrized with mean* μ *and variance* $2\sigma^2$ *is defined by*

$$\Pr[\mathrm{Lap}(\mu,\sigma)=x]=\frac{1}{2\sigma}e^{-\frac{|x-\mu|}{\sigma}}$$

In order to describe the mechanism we first need to introduce the following notation:

Definition 5.5 (Sensitivity) *The* sensitivity *of a function* f *is defined as*

$$\max_{\text{neighbouring}D,D'} |f(D)-f(D')| \ ,$$

i.e., the sensitivity of a function is defined by looking at the maximum difference (in absolute value) between the output of the function computed on two neighbouring databases (note that the definition of sensitivity relies in turn on the definition of neighbouring databases therefore the multiple definitions of neighbouring databases present in the literature give rise to multiple definitions of sensitivity).

Construction 5.6 (Laplace Mechanism) *We can construct a mechanism* $M(D)$ *for computing a function* f *with sensitivity* s *on input a database* D *with* ε*-differential privacy as follows:*

1. *Compute* $y=f(D)$*;*
2. *Compute* $e=\mathrm{Lap}(\frac{s}{\varepsilon})$*;*
3. *Output* $z=y+e$*;*

The Laplace mechanism adds a significant amount of noise to the output of the computation. While it is hard to draw general conclusions it can be said that the Laplace mechanism produces acceptable results for functions with small sensitivity (counting, histogram, etc.), but very poor performances for functions with high (or even unbounded) sensitivity. As an example, consider the setting in which one wants to compute the average wealth of a population. In principle there is no bound to what an individual's wealth might be, thus the sensitivity of the function is unbounded. A possible approach is therefore to find a range $[min,max]$ of plausible values, and then let the inputs who exceed the bound "saturate" on the bounds. While on one hand one might want to minimize the gap $max-min$ to reduce the amount of noise added by the Laplace mechanism, one has also to consider the error added to the computation by truncating the inputs.

On the positive side, the simplicity of the Laplace mechanism implies that it relatively easy to implement this in combination with MPC.

5.4.2 Sample-And-Aggregate

The *sample-and-aggregate* mechanism is a variation on the Laplace mechanism that was introduced in [140]. In a nutshell, this mechanism splits the input database into smaller subsets, evaluates the function on the subsets and, finally, recombines the result adding some noise. This has been proven to provide better efficiency. A high level overview of the mechanism is given.

Construction 5.7 (Sample and Aggregate) *Given parameter B for the block size, boundaries min and max for the input values, and a block function g we can construct a mechanism $M(D)$ for computing a function f with sensitivity s on input a database D with ε-differential privacy as follows.*

1. *Randomly assign the elements of the database D into $m = |D|/B$ databases $D_1, \ldots, D_m$ of size B each.*
2. *Compute $y_i = g(D_i)$ for all $i = 1, \ldots, m$;*
3. *If $y_i < min$ then $y_i = min$ for all $i = 1, \ldots, m$;*
4. *If $y_i > max$ then $y_i = max$ for all $i = 1, \ldots, m$;*
5. *Compute $e = \mathrm{Lap}(\frac{max-min}{m \cdot \varepsilon})$;*
6. *Output $z = \frac{1}{m} \sum_1^m y_i + e$;*

While more complex than the simple Laplace mechanism, this mechanism still uses (for the most part) linear operations which can be efficiently implemented in MPC, as well as comparisons which, while typically a more expensive, still guarantee reasonable performance.

5.4.3 The Exponential Mechanism

The exponential mechanism was introduced in [130]. Its implementation is much more cumbersome than the previously introduced mechanisms, but it provides more accurate results. Given a database D and candidate output z for the function $y = f(D)$, we define a scoring function $u(D,z)$ which measures the quality of the output z (for instance, the difference between z and y). Then, if we call s the sensitivity of f we can describe the exponential mechanism as follows:

Construction 5.8 (Exponential Mechanism) *Given a function f with sensitivity s and scoring function u, and given a privacy parameter ε the exponential mechanism randomly samples an output z according to the following probability distribution:*

$$\Pr[M(D) = z] = \frac{exp(\frac{\varepsilon}{2s} \cdot u(D,z))}{\sum_{z'} exp(\frac{\varepsilon}{2s} u(D,z'))}$$

where $exp(x) = e^x$.

That is, the exponential mechanism assigns higher probability to higher-quality outputs. Unfortunately, due to its (in)efficiency properties, the exponential mechanism does not appear to be easily combined with known MPC techniques.

5.5 Combining Differential Privacy and MPC

There is relatively little previous work in combining MPC and Differential Privacy. We briefly review here some of the work which is mostly relevant for our project.

Randomized Responses. The simplest multi-party differentially private mechanism is the *randomized response* mechanism which can be summarized as follows: Assume for simplicity that the input of each party is a bit. Each party randomizes their inputs (e.g., flips their bit or adds noise with the necessary distribution required to achieve differential privacy). Then, all the *noisy* versions of the inputs are publicly released, and the target function is computed using these noisy inputs. This mechanism has been known since the 60s [175] and its properties have been recently re-evaluated in the light of modern developments in [108]. This mechanism is also known in the literature as the *local model* or the *input perturbation* mechanism.

Combining How and What. In [16] the authors consider the problem of combining DP mechanisms with MPC protocols. The straightforward combination of MPC with DP is to choose an adequate DP mechanism (*what* to compute) and then combine it with an appropriate MPC protocol (*how* to compute). The main question addressed by this work is whether there are benefits in choosing the MPC protocol and the DP mechanism *at the same time*. The main result of the paper is that (for a collection of interesting functions e.g., computing the sum of the inputs) there is no benefit in the combined approach when the required noise is low. On the other hand, for large amount of noises, the combination of MPC and DP already in the design phase produces more efficient mechanisms. In particular, the *randomized response* mechanism described above provides an efficient solution both from an MPC and a DP point of view. The main intuition is that, when the noise to be added to the result is small, the local model does not achieve the necessary level of privacy (and therefore MPC can help by hiding the parties inputs). For large values of noise, the perturbation of the inputs guarantees enough privacy and no (expensive) MPC protocols need to be run.

Sharemind. In [148] an attempt at combining differential privacy and MPC is considered. In this paper the authors consider the Sharemind framework for MPC (see Section 6) and implement on top of that several variants of the sample-and-aggregate mechanism described before. The main conclusions of this work is that due to the additive nature of the mechanism (a function is computed and then noise is added) the efficiency of the mechanism is comparable (less than twice as slow) to a non-differentially private version. Great care is taken in choosing MPC building blocks that make the overall performance as good as possible, and to use their privacy budget in the best possible way.

Unfortunately, the paper does not go in depth in the way that they deal with the fact that the computation requires non-integer operations required in the generation of a Laplace random value. We will describe the issue with this in the next subsection.

ODO Mechanism. In [66] the authors propose MPC protocols secure against a minority ($t \approx n/3$) of actively corrupted parties for evaluating a differentially private mechanism. Their main technical contribution are MPC protocols to jointly generate noise according to the Binomial and Poisson distribution, that can be used in turn to approximate the noise distributions required for implementing differentially private mechanisms.

DStress. In [144], Papadimitriou et al. develop DStress, a framework for privacy-preserving, distributed graph analytics. DStress targets a setting in which knowledge of a graph is distributed across multiple parties, i.e., each party only knows a private *sub-graph*; the parties wish to compute a query over the entire graph while keeping the individual sub-graphs private. To this end, DStress combines the use of MPC and differential privacy. Notably, DStress leverages differential privacy in two ways: (i) to provide output privacy, but also (ii) to trade off security for scalability. We further elaborate on

the latter. The authors consider a use case that requires MPC to run across a large number of parties ($\sim$100); however, current general-purpose MPC solutions do not scale beyond $\sim$10 parties[2]. To this end, the authors develop a special-purpose protocol which, at a high-level, requires that intermediate values are revealed throughout the computation. To address the incurred leakage, the values are noised prior to being revealed, using the *geometric mechanism* [84], a discrete version of the Laplace mechanism described above. Therefore, instead of producing zero leakage (as with standard MPC), the execution of the computation itself is ε-differentially private. The protocol is secure against a minority of passively-corrupted parties, and operates on integers only. The authors evaluate their system on workloads for measuring systemic risk in financial networks and demonstrate practical performance for 100 parties and a corruption threshold of 20.

5.6 Challenges and Research Directions

Differential Private MPC with Dishonest Majority and Active Security. To the best of our knowledge no previous work considers the implementation of differentially private mechanism in the presence of a dishonest majority and while guaranteeing security. This is a very interesting research direction, since it addresses the most comprehensive threat model. At the same time, it seems technically challenging: the presence of a dishonest majority immediately prevents solutions which guarantee *output delivery* and *fairness*. In particular, it will be challenging to design protocols which are resilient to malicious parties aborting as a function of the generated noise (and therefore biasing the distribution).

Efficient Implementation of Differentially Private Mechanism. Dealing with DP in MPC requires dealing with non-integer types, as can be seen from the description of the different differentially private mechanisms described above. Unfortunately, known efficient MPC protocols only allow to perform computation with finite precision. Therefore, novel efficient approaches for implementing differentially private mechanism will have to be found as part of this project.

Modelling and Dealing with Approximations in MPC. As we have seen differentially private computations are necessarily noisy and approximated. While natural MPC protocols only support additions and multiplications of (modular) integers (the so called *arithmetic black-box*), implementations of approximated, non-integer types such as *fixed-point* and *floating-point* operations have been proposed, see for instance [61] and reference therein. Such implementations necessarily use truncations and approximation as a tool for keeping the representation of the internal values from growing outside of the allowed boundaries. Unfortunately we have found strong evidence that such operations can break the security of cryptographic protocols, even to the point which is possible to break even the relatively weak notion of *input-indistinguishability*.

To the best of our knowledge, the only paper ever addressing the issue of privacy leakages due to approximation in MPC protocols is [70]. Their solution is, in a nutshell, to approximate a computation to a higher level of precision and then to add enough noise so as to mask the error due to rounding. Unfortunately this approach cannot be applied to existing MPC protocol which follow the arithmetic black-box paradigm, since these protocols *do not* allow for arbitrary precision.

[2]The authors further reject solutions such a Sharemind where the computation is delegated to a few select parties since this approach does to provide the desired (absolute) corruption threshold.

6 Implementations

In this section we survey existing MPC software frameworks. We focus on *general-purpose* frameworks, i.e., frameworks that support the evaluation of any computable function, as opposed to frameworks that only support specific protocols such as private set intersection.

Malkhi et al. implemented the first general-purpose MPC software platform, Fairplay, in 2004 [128]. Since then, the community has produced a wide array of frameworks [59, 26, 80, 112, 152, 120] and significantly advanced the maturity of MPC technology [8]. Much of the work following Fairplay focuses on improving runtime performance and has yielded immense progress [8]. Furthermore, the past decade has seen several successful production deployments of MPC, including an implementation of secure auctions [28], tax fraud detection [23], and federated learning [29]. While open-source research prototypes comprise a majority of the MPC frameworks available today [59, 80, 112, 152], several closed-source commercial platforms exist as well [26, 120].

The two prominent cryptographic constructs upon which MPC frameworks are built are garbled circuits [179] (see Section 2.1.3) and linear secret-sharing [159] (with [86, 17, 39] pioneering the approach). We discuss both garbled-circuit based frameworks [182, 124] and linear-secret sharing based frameworks [26, 80, 112, 152]. Furthermore, we discuss ABY [59], a *mixed-protocol* framework which combines the use of both techniques.

Next, we describe in more detail a selection of existing MPC frameworks. The list is not comprehensive but rather meant to provide a general overview of the different directions in which MPC software framework research and development has evolved over the past years. For each framework we summarize its security guarantees, implementation details, programming interface, and applications (both commercial and within the research community). For an in-depth cross-examination on the performance and maturity of currently available MPC frameworks we refer the reader to a recent survey by Archer et al. [8].

6.1 VIFF

VIFF [80] (also see Section 3.1.2) was used as the MPC backend in the Danish sugar beet auction [28] in 2008[3], and as such the first MPC framework to see production deployment. VIFF is implemented as a Python module. The framework makes use of operator overloading to allow programmers to write high-level programs consisting of standard operations such as addition and multiplication. These high-level programs are agnostic to the underlying MPC backend protocols used to execute them. VIFF provides several backend protocols–referred to as *runtimes*–making it flexible both in the number of parties (2 to n) it supports as well as its security guarantees (passive or active, with varying corruption thresholds). VIFF uses an asynchronous execution model to improve performance. Asynchronous communication and execution are built on top of the networking engine Twisted. While the original version of VIFF has since been retired (http://viff.dk/), a new version of the framework has recently been released (http://www.win.tue.nl/ berry/TUeVIFF/), improving both performance and usability.

Security

VIFF supports four runtimes with different security guarantees:

- A passively secure runtime for $n > 2$ parties and corruption threshold $t < \frac{n}{2}$ based on linear secret sharing

[3]The auction still runs every year and VIFF remains the MPC backend.

- A passively secure runtime for 2-party computation based on the Paillier cryptosystem [142]
- An actively secure runtime for $n > 2$ parties and corruption threshold $t < \frac{n}{3}$ described in [48]
- An actively secure runtime for 2-party computation based on [139]

Implementation and Programming Interface

VIFF is implemented in Python on top of the networking engine Twisted. The most recent version is available at (http://www.win.tue.nl/ berry/TUeVIFF/). VIFF is a Python module; a programmer can thus access all its functionality through the module's API and further leverage all of the other Python language features. Operator overloading allows a programmer to naturally specify circuits consisting of arithmetic and comparison operators.

Applications

Apart from the sugar beet auction deployment [28], more recently, De Hoogh et al. used VIFF to implement decision tree learning [55]; Schoenmakers et al. used VIFF as a backend for a framework for privacy-preserving verifiable computation [158].

6.2 Sharemind

Sharemind [26] (also see Section 3.1.1) is a commercial MPC platform. Since its conception in 2007 [22], it has been used in several production deployments [23, 24]. The core Sharemind runtime environment is built for performance; it uses a three-party protocol based on additive secret-sharing in the ring $\mathbb{Z}_{2^{32}}$ (alternatively $\mathbb{Z}_{2^{64}}$). The protocol is secure against a passive adversary able to corrupt at most one party (honest-majority). Protocols secure against a semi-honest corrupted majority require the use of slow operations based on public-key cryptography [42]; the Sharemind threat model on the other hand allows the authors to implement secure arithmetic using only highly-efficient, information-theoretically secure constructs. Furthermore, operating in $\mathbb{Z}_{2^{32}}$ removes the overhead of modular reductions (in contrast to classical protocols such as used in [80] that operate in a prime field) since it can be implemented "for-free" leveraging 32-bit arithmetic wrap-around.

Sharemind differs from preceding MPC frameworks in its deployment model by distinguishing between *input parties*, *compute parties*, and *result parties*. *Input parties* provide input into a given computation but do not necessarily participate further in its evaluation. They do so by secret-sharing their data with the *compute parties*. The *compute parties* use the Sharemind runtime environment to perform the computation; as a result a deployment requires exactly three compute parties. Upon completion, the compute parties publish the shares of the result to a set of *result parties*. This deployment model allows Sharemind to support an arbitrary number of input and result parties while enjoying the performance benefits of the specialized three-party protocol described above.

Sharemind provides a C-like, domain-specific language [102] for application development, and an R-like front-end for privacy-preserving statistical analysis [25]. Recent efforts extend Sharemind with support for 2-party computation [162], as well as covert and active security [143].

Security

The core Sharemind runtime provides a three-party protocol secure against a semi-honest adversary capable to corrupting at most one compute party. Extensions (at the cost of performance) to covert and active security are outlined in [143].

Implementation and Programming Interface

The runtime environment along with the front-end language SecreC are implemented in C++. Most of the higher-level constructs such as oblivious sorting are implemented in SecreC itself. Licenses are required for both academic and commercial use though an emulator of the runtime is publicly available at https://sharemind.cyber.ee/secure-computing-platform/#sdk. Programs are implemented in SecreC [102], a C-like domain specific language. SecreC supports basic arithmetic operations as well as a wide range of higher-level constructs, ranging from various data structure operations, e.g., sorting, to statistical operations. *Vectorized operations* are a crucial component of the language as they allow a programmer to specify operations over vectors which will be executed in parallel, using batched communication. This has a significant impact on performance for communication-heavy operations such as comparisons [26].

Applications

Sharemind has been used in several production deployments [24, 23, 27], most recently to build upon a previous deployment of a tax fraud detection system [27] in a cloud environment. This constitutes the largest MPC deployment to-date. A number of MPC research efforts leverage Sharemind as an MPC backend, see https://sharemind.cyber.ee/research/.

6.3 Bristol-SPDZ

The framework by Keller et al. [112] is an implementation of SPDZ. SPDZ designates a family of actively-secure, dishonest-majority MPC protocols [49, 50, 52] that operate in the pre-processing model, i.e., separate a computation into two phases, a slow, yet data-independent *offline* phase, and a fast *online* phase. The framework provides two components, (i) a highly-efficient runtime environment for the online phase of SPDZ, and (ii) an implementation of the offline phase based on MASCOT [109]. We focus our discussion on the former. Since SPDZ is an arithmetic-sharing based scheme, the runtime environment is implemented as a virtual machine that operates on additive secret shares (as well as public values). It can perform linear combinations, open shares, and consume material generated during the offline phase. This stands in contrast to most other arithmetic-sharing based frameworks such as [26, 80]: normally additions and multiplications comprise the primitive operations. Building up from linear combination, open, and consume operations instead enables additional optimization both in how higher-level constructs such as comparison are implemented, as well as in minimizing communication rounds between the parties. The framework comes with an embedded, domain-specific language for specifying programs and a compiler for converting these into linear-combination, open, and consume operations.

Security

The framework supports $n \geq 2$ parties and is secure against an active adversary capable of corrupting up to $n-1$ parties.

Implementation and Programming Interface

The virtual machine, i.e., execution environment is implemented in C++ whereas the compiler is implemented in Python. The front-end language is embedded in Python, giving the programmer full access to Python language features at compile-time. The domain-specific language ex-

poses basic arithmetic and boolean operators, comparisons, and higher-level operations on data-structures such as sorting, shuffling, and shortest-path. The framework is open-source and available at https://github.com/bristolcrypto/SPDZ-2.

Applications

Keller et al. subsequently use their framework to implement and evaluate (i) various oblivious RAM schemes and oblivious data structures in the multi-party setting in [111], (ii) symmetric-key primitives that are more amenable to the MPC setting than standard primitives in [92].

6.4 Wysteria

The Wysteria programming language [152] and associated runtime environment enable the development and execution of *hybrid-mode*[4] MPC programs. Hybrid-mode programs consist of *local* stages which the participating parties perform locally over their own data, and *aggregate* stages in which the parties use MPC to process their combined data. For certain computations, this mode of operation can yield significant performance improvements [113]. Wysteria offers a formal type system and, consequently, automatic type checking. While other hybrid-mode frameworks exist [80], most are implemented as libraries in a general-purpose language; local stages must therefore be implemented directly in the host language while MPC functionality is invoked through the given framework's API. In contrast, Wysteria has a single, unified interface for specifying both local and aggregate stages. Wysteria programs are compiled into boolean circuits and executed in the framework by Choi et al. [41] which implements the GMW protocol (see Section 2.2.2).

Security

Wysteria supports $n \geq 2$ parties and is secure against a semi-honest adversary capable of corrupting up to $n-1$ parties.

Implementation and Programming Interface

The Wysteria language and compiler are implemented in OCaml; the framework by Choi et al. [41] serves as the MPC backend. The authors leverage the Z3 SMT solver [58] for formal verification. The framework is open-source and available at https://bitbucket.org/aseemr/wysteria/wiki/Home. The front-end language is a functional domain-specific language with support for basic arithmetic operations and comparison. A programmer specifies local steps for each party as well as aggregate MPC steps across parties.

Applications

Wysteria is used as a backend for DStress [145], a framework for privacy-preserving, distributed graph analytics (see Section 5.5).

[4] We note that the original work refers to such programs as mixed-mode. Unfortunately, this term is easily mixed up (no pun intended) with the term *mixed-protocol* which refers to frameworks such as ABY [59]–frameworks that allow for the mixed use of different MPC schemes. We use *hybrid-mode* to disambiguate.

6.5 ABY

ABY (Arithmetic-Boolean-Yao) [59] is a *mixed-protocol* framework. In contrast to frameworks such as VIFF [80], mixed-protocol frameworks support the *combined use of multiple MPC schemes*; ABY supports computation over arithmetic secret-shares, boolean secret-shares, and Yao's garbled circuits along with conversion protocols between the three modes. A programmer may thus compose the use of all three schemes within a single computation. For certain computational tasks, this approach yields significant performance gains (ABY demonstrates a 13-fold speed-up over using a single scheme for private-set intersection); this gain is due to the fact that the performance of standard operations varies greatly across schemes–multiplications, for instance, are more efficient over arithmetic shares, while garbled circuits perform better on comparisons. Conversion between schemes does not come for free however and has been shown to be a performance bottle neck for mixed-protocol frameworks [114]; to this end ABY provides novel conversion protocols based on precomputed oblivious transfer extensions [15, 9], greatly improving performance. In ABY, it is up to the programmer to specify the mode of operation within a computation; conversions between schemes are invoked manually. This leaves the developer with the highest degree of freedom to optimize a particular implementation but also requires expert knowledge [114]. The authors note that ABY can be combined with existing work on automatically detecting the optimal composition of schemes for a particular computation [114] to increase accessibility.

Security

ABY supports 2-party computation; its underlying schemes as well as conversion protocols between them are secure against a semi-honest adversary.

Implementation and Programming Interface

ABY is implemented in C++, and available at https://github.com/encryptogroup/ABY. ABY allows the programmer to specify the circuit to be evaluated (and which scheme to use) using basic arithmetic, boolean, and comparison gates, as well as conversion gates. This functionality is exposed as a C++ library.

Applications

Pinkas et al. use ABY to benchmark private-set intersection implementations in [151]; Liu et al. do so in their work on oblivious neural networks [125].

6.6 ObliVM

The ObliVM MPC framework [124] aims to ease the development process of MPC applications. ObliVM consists of three components: (i) the domain-specific language ObliV-lang which offers high-level language abstractions common to general purpose programming languages, such as control flow and data structures, (ii) a compiler which translates ObliV-lang programs into *trace-oblivious* programs (programs with control-flow independent of secret values), and consequently circuits, using special-purpose optimized oblivious algorithms when possible and falling back onto generic ORAM to realize trace-obliviousness otherwise, and (iii) a Yao's garbled circuits-based MPC backend (see Section 2.1.3) for evaluating the circuits.

Security

ObliVM supports 2-party computation with passive security.

Implementation and Programming Interface

ObliVM is implemented in Java. ObliVM distinguishes between two types of end-users: *non-specialist programmers* and *expert programmers*. Non-specialist programmers may use the C-like ObliVM-lang DSL and its high-level abstractions, e.g., *if-statements*, to develop applications. Expert programmers can develop new user-facing abstractions and efficient oblivious algorithms and integrate alternative MPC backends into ObliVM. The project is open source and available at http://www.oblivm.com/.

Applications

ObliVM has found application in numerous research efforts. The follow-up effort GraphSC [136] by Nayak et al. extends ObliVM with language abstractions and oblivious algorithms for graph processing and automates their parallelization. In [101] Jagadeesan et al. use ObliVM as a backend for a decentralized software defined networking solution. Wagner et al. implement privacy-preserving microbiome analysis atop of ObliVM in [171]. More recently, Bater et al. build a framework for multi-party database querying in [12]. Bater et al.'s solution automatically splits queries into local steps that can be performed directly on each database and aggregate steps which use MPC to combine the results of the local steps; here ObliVM serves as the MPC backend.

6.7 Obliv-C

Obliv-C, developed by researches at University of Virginia [182], is a domain specific language for hybrid-mode MPC programs. Obliv-C extends the C programming language with so-called *oblivious* versions of the basic C data types on which all operations are done securely using MPC. This way Obliv-C makes it possible to embed code to be evaluated using MPC into a regular C program, which can take advantage of the features and libraries of C when doing local (non-secure) computation.

Obliv-C focuses on being extensible; it makes it easy to implement libraries of highly optimized MPC functionality, which can then be reused in various Obliv-C programs. The *Absentminded Crypto Kit* project by Doerner https://bitbucket.org/jackdoerner/absentminded-crypto-kit collects such a library offering various functionality including big interger math, various hash functions and symmetric ciphers, *Stable Matching* algorithms [63] and even a number of different ORAM implementations for MPC [172, 62, 184].

Additionally, the MPC protocol run by Obliv-C programs at runtime is not fixed. Namely, Obliv-C enables developers to easily write implementations of new protocols, provided these work in the boolean setting. Switching the MPC protocol used by an Obliv-C program then only requires changing a single line of the program.

Security

As mentioned above Obliv-C potentially supports any MPC protocol in the Boolean setting. Currently, two variations of the two-party garbled circuit protocol are implemented. One variant secure against semi-honest adversaries, including various optimizations such as the *half-gate* technique (see Section 2.1.3). The other variant is based on the *dual execution* technique of Huang *et al.*, which gives a very efficient protocol secure against malicious adversaries at the cost of leaking a single bit.

Implementation and Programming Interface

As described above Obliv-C is an extension of the C programming language. The added syntax, however, is rather minimal. Essentially, Obliv-C only introduces a single new keyword `obliv` which is used to indicate the oblivious data types which computation must be done using MPC. A special feature of the Obliv-C language are oblivious *if-statements* where the condition may be an oblivious Boolean.

Compiling an Obliv-C program works by running the Obliv-C code through a precompilation step which translates the program into regular C program. This is done by translating operations on the oblivious data types into function calls. The oblivious if-statements are handled essentially by securely computing both branches of the statement and translating any assignment into conditional assignments. The resulting C program can then be compiled by a regular C compiler such as *gcc*.

Obliv-C open-source and available at https://oblivc.org/.

Applications

A number of interesting applications have been implementing using the Obliv-C language. For example, Doerner et al. reports on a scalable implementation of the Gale-Shapely algorithm for the stable matching problem [63]. Gascón et al., describe an implementation of linear regression on high dimensional data and Tian et al. describe an implementation to securely aggregate multiple locally trained machine learning models using MPC [167]. Additionally, a commercial product SECCOMP (https://www.calctopia.com/) uses Obliv-C to implement a secure distributed spreadsheet solution based on MPC.

6.8 FRESCO

FRESCO is a hybrid-mode MPC framework implemented as a Java library. FRESCO offers a high degree of modularity by decoupling *functionality specification*–the description of the circuit to be evaluated–from *execution*, i.e., the MPC backend scheme (whether it be boolean or arithmetic) used to evaluate the circuit. Functionality is specified in terms of basic arithmetic and boolean operators; more complex protocols such as comparison are constructed from these. This allows for the desired functionality to be executed in any MPC backend that supports the basic operators. MPC backends may however also implement specific, optimized protocols for higher-level functionality. FRESCO currently provides an arithmetic backend suite implementing SPDZ [49, 50, 52], and a boolean backend suite based on TinyTable [51], an actively secure two-party scheme. Furthermore, FRESCO allows a programmer to explicitly specify portions of the circuit which may be executed in parallel; MPC backend schemes may provide optimized evaluation strategies for these sub-circuits.

Security

FRESCO provides two backend suites:

- An arithmetic suite based on SPDZ [49, 50, 52] that supports $n \geq 2$ parties and is secure against an active adversary capable of corrupting up to $n-1$ parties.
- A boolean actively-secure two-party suite based on TinyTable [51].

Implementation and Programming Interface

FRESCO is implemented in Java; FRESCO functionality is consequently exposed as a Java library. A programmer has access to basic arithmetic operations as well as higher-level operators ranging from comparison to statistical operations. FRESCO is open-source and available at https://github.com/aicis/fresco.

Applications

FRESCO was used as an MPC backend in a prototype deployment for financial benchmarking in [47].

6.9 SCAPI

SCAPI (Secure Computation API) [68] is a library for developing MPC frameworks and applications. Unlike the MPC frameworks described previously, SCAPI does not provide a high-level programming API and instead exposes to the end-user low-level functionality that is commonly required for MPC, ranging from system-level features such as networking, to cryptographic primitives, such as oblivious transfer.

Security

Apart from providing a number of low-level cryptographic primitives, SCAPI comes with passively- and actively-secure instantiations of the Yao garbled-circuits protocol, supporting two parties (see Section 2.1.3).

Implementation and Programming Interface

SCAPI consists of two libraries, a Java library and a C++ library. The system-level and cryptographic primitives are exposed through the associated library APIs. To evaluate a circuit using the Yao garbled-circuits implementations, the circuit must first be encoded in an external file (following a format akin to https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/). SCAPI is open-source and available at https://github.com/cryptobiu/scapi.

Applications

A number of research efforts leverage SCAPI as a backend. For instance, Jawurek develop and implement a Yao garbled-circuits based protocol for generic zero-knowledge proofs in [103]; the authors implement oblivious transfer in SCAPI. More recently, Kolman and Pinkas use SCAPI to prototype a privacy-preserving impersenation detection system in [118]. Campanelli et al. use SCAPI in their work on protocols for fair exchange in [36].

7 Conclusion

Secure multi-party computation (MPC) can be seen as a tool that allows a number of distributed entities to jointly compute a functionality on their private data securely, even if some of the parties involved are dishonest. As we have discussed, the fact that it can be used to securely evaluate *any* functionality means that there are a myriad of interesting scenarios where one can use MPC. The main question is how practical the MPC-based solutions are, and whether or not this can be improved, without compromising the desired security properties.

WP1 of the SODA project aims to improve upon existing work in MPC such that it can be later used in practice, in accordance to the SODA project goals. To aid in this goal, this document gives a state of the art analysis of MPC and its relevance to other areas such as oblivious RAM, oblivious data structures, and differential privacy. From our analysis, we can make the following conclusions.

Basic Considerations. The first thing to consider is how we want to model our secure functionality. The main ones used are Boolean or arithmetic circuits, but some recent work have investigated the use of fixed/floating point arithmetic and oblivious RAM.

Moreover, we also have to decide if we want passive or active security. A multi-party protocol can first constructed with passive adversaries in mind, then *compiled* into an actively secure one with a constant overhead (say, 10 times slower than the passively secure protocol). In this sense, one can just focus on constructing efficient passively secure MPC, then use such a compiler to get active security. However, there exists cases where a multi-party protocol is much more efficient if we aim for active security from the start, and circumvent the overhead of using a compiler.

Two-Party Computation. As discussed in Section 2, two-party computation is done efficiently using Yao's garbled circuits or GMW-style secret sharing. Computation-wise, there is not much difference between the two. Yao has a constant number of rounds (regardless of the circuit size) but high communication in total, whereas GMW has several rounds with a smaller amount of bandwidth, but the large number of rounds which makes the constructions latency-dependent. Most of the computation (and communication) in GMW can be moved to the preprocessing phase, making the online phase very efficient.

Multi-party Computation. If we have three or more parties, we can talk about the notion of an honest majority. For three parties with at most one corruption, there are many efficient constructions, the most notable of which are Sharemind and Viff, which use secret sharing over a large field. Recent work has also been done which use secret sharing over $\mathbb{F}_2$.

For more than three parties and/or a dishonest majority, most of the efficient constructions also use secret sharing, with active security guaranteed using information-theoretic MACs. The bulk of the effort is done in the preprocessing phase in the form of generation of many multiplication triples. Generating multiplication triples efficiently can be done either by using homomorphic encryption, such as the case of SPDZ, or by using OT, such as the case of MASCOT. An alternative to using secret sharing is a generalization of garbled circuits in the multi-party setting, which can be done efficiently using TinyOT plus authenticated garbling.

Data-Oblivious MPC. RAM programs have the advantage that it models the workings of a real world program, and its size is not dependent on the function to be computed or the size of the input. However, as discussed in Section 4, it is a challenge to make RAM access *oblivious* and MPC-

friendly. The main things to consider are the storage, online, and bandwidth overheads. Some ORAM constructions are optimized for different MPC techniques. For instance, Circuit-ORAM is highly compatible with garbled circuits.

To make MPC practical, it is important to have efficient MPC-based implementations of common abstract data types, which result in *oblivious data structures*. The main considerations here include what is leaked in each data structure, how data is partitioned, and what MPC techniques are compatible. To get efficiency, we also want solutions that support parallelization.

Differential Privacy. One of the main goals of the SODA project is to perform secure big data analytics in health systems. As we have seen in Section 5, MPC alone might not be sufficient to achieve the necessary level of privacy for individuals, since the output of the computation necessarily reveals information about their data. Instead, we need techniques in *differential privacy*, which aims to provide accurate computation on database elements without giving information about individual database entries.

Combining differential privacy and MPC techniques seems to be the way forward to perform secure big data analytics. However, this is far from straightforward to achieve, in particular due to the fact that using differential privacy in MPC requires being able to deal with non-integer types which, depending on representation, does not necessarily lead to secure protocols. How to model and deal with approximations in MPC is among some of the research directions we aim to pursue in WP1.

Implementations. Section 6 showed that MPC is not just a theoretical concept by reviewing the several general-purpose software frameworks for MPC, i.e., frameworks that can evaluate any computable function. Some of the best alternatives include VIFF, Sharemind, Bristol-SPDZ, Wysteria, ABY, OblivVM, Obliv-C, FRESCO and SCAPI.

These frameworks have been used in practice. For example, VIFF was used in a Danish sugar beet auction in 2008 while Sharemind is used in a tax fraud detection system with data volumes the size of Estonia's economy. Choosing the best framework to use depends on several factors, such as preferred programming language, what function you wish to implement, and how you want to model the function. In SODA, we continue to develop the FRESCO framework, taking into account lessons from recent developments in other frameworks.

References

[1] *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[2] *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015.

[3] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the k th-ranked element. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 40–55. Springer, 2004.

[4] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, volume 7859 of *Lecture Notes in Computer Science*, pages 239–257. Springer, 2013.

[5] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In Hugo Krawczyk, editor, *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, volume 8383 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 2014.

[6] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 843–862, 2017.

[7] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817, 2016.

[8] David W Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. *IEEE Security & Privacy*, 14(5):48–56, 2016.

[9] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548. ACM, 2013.

[10] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 271–286, 2015.

[11] Michael Barbaro, Tom Zeller, and Saul Hansell. A face is exposed for aol searcher no. 4417749. *New York Times*, 9(2008):8For, 2006.

[12] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6):673–684, 2017.

[13] Carsten Baum, Ivan Damgård, Kasper Green Larsen, and Michael Nielsen. How to prove knowledge of small secrets. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, volume 9816 of *Lecture Notes in Computer Science*, pages 478–498. Springer, 2016.

[14] D Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology - CRYPTO 1991*, 1992.

[15] Donald Beaver. Precomputing oblivious transfer. *Advances in Cryptology CRYPTO 95*, pages 97–109, 1995.

[16] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 451–468. Springer, 2008.

[17] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 1–10, New York, 1988. ACM.

[18] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

[19] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* [2], pages 287–304.

[20] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[21] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[22] Dan Bogdanov. How to securely perform computations on secret-shared data. Master's thesis, Institute of Computer Science, University of Tartu, 2007.

[23] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International Conference on Financial Cryptography and Data Security*, pages 227–234. Springer, 2015.

[24] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings on Privacy Enhancing Technologies*, 2016(3):117–135, 2016.

[25] Dan Bogdanov, Liina Kamm, Swen Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[26] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. *Computer Security-ESORICS 2008*, pages 192–206, 2008.

[27] Dan Bogdanov, Marko J oemets, Sander Siim, and Meril Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Technical Report T-4-24, Cybernetica AS, http://research.cyber.ee/., 2016.

[28] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography*, volume 5628, pages 325–343. Springer, 2009.

[29] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy preserving machine learning. *IACR Cryptology ePrint Archive*, 2017:281, 2017.

[30] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious ram practical. *URL http://dspace. mit. edu/handle/1721.1/62006. Technical Report*, 2011.

[31] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Kushilevitz and Malkin [119], pages 175–204.

[32] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In Sudan [165], pages 357–368.

[33] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.

[34] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/.

[35] Niklas Büscher and Stefan Katzenbeisser. Faster secure computation through automatic parallelization. In Jung and Holz [107], pages 531–546.

[36] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. *Commun. ACM*, 2017.

[37] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *Theory*

of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II, volume 9986 of *Lecture Notes in Computer Science*, pages 61–90, 2016.

[38] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In Sudan [165], pages 169–178.

[39] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 11–19, New York, 1988. ACM.

[40] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In Kushilevitz and Malkin [119], pages 205–234.

[41] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA*, volume 7178, pages 416–432. Springer, 2012.

[42] Benny Chor and Eyal Kushilevitz. A zero-one law for boolean privacy. *SIAM Journal on Discrete Mathematics*, 4(1):36–47, 1991.

[43] Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[44] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.

[45] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 342–362, 2005.

[46] Ronald Cramer, Ivan Damgård, Chaoping Xing, and Chen Yuan. Amortized complexity of zero-knowledge proofs revisited: Achieving linear soundness slack. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 479–500, 2017.

[47] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *International Conference on Financial Cryptography and Data Security*, pages 169–187. Springer, 2016.

[48] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography*, volume 5443, pages 160–179. Springer, 2009.

[49] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. In *SCN*, volume 7485, pages 241–263. Springer, 2012.

[50] 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, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[51] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Annual International Cryptology Conference*, pages 167–187. Springer, 2017.

[52] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[53] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641. Springer, 2013.

[54] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming*. PhD thesis, Eindhoven University of Technology, 2012.

[55] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. Practical secure decision tree learning in a teletreatment application. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2014.

[56] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. *Universally Verifiable Outsourcing and Application to Linear Programming*, volume 13 of *Cryptology and Information Security Series*, chapter 10. IOS Press, 2015.

[57] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, volume 9646 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2016.

[58] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[59] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[60] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Kushilevitz and Malkin [119], pages 145–174.

[61] Vassil Dimitrov, Liisi Kerik, Toomas Krips, Jaak Randmets, and Jan Willemson. Alternative implementations of secure real numbers. In Weippl et al. [177], pages 553–564.

[62] Jack Doerner and abhi shelat. Scaling oram for secure computation. In *To appear CCS'2017*, 2017.

[63] Jack Doerner, David Evans, et al. Secure stable matching at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1602–1613. ACM, 2016.

[64] Jack Doerner, David Evans, and Abhi Shelat. Secure stable matching at scale. Cryptology ePrint Archive, Report 2016/861, 2016.

[65] W. Du and M. Atallah. Protocols for secure remote database access with approximate matching. In A. K. Ghosh, editor, *E-Commerce Security and Privacy*, pages 87–111. Springer, 2000.

[66] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Eurocrypt*, volume 4004, pages 486–503. Springer, 2006.

[67] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2006.

[68] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. *IACR Cryptology EPrint Archive*, 2012:629, 2012.

[69] Prastudy Fauzi, Helger Lipmaa, and Bingsheng Zhang. Efficient modular NIZK arguments from shift and product. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, November 20-22. 2013. Proceedings*, volume 8257 of *Lecture Notes in Computer Science*, pages 92–121. Springer, 2013.

[70] Joan Feigenbaum, Yuval Ishai, Tal Malkin, Kobbi Nissim, Martin J. Strauss, and Rebecca N. Wright. Secure multiparty computation of approximations. *ACM Trans. Algorithms*, 2(3):435–472, 2006.

[71] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In Weippl et al. [177], pages 1304–1316.

[72] Matthew Franklin, Mark Gondree, and Payman Mohassel. Multi-party indirect indexing and applications. In *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'07, pages 283–297, Berlin, Heidelberg, 2007. Springer-Verlag.

[73] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. Tinylego: An interactive garbling scheme for maliciously secure two-party computation. *IACR Cryptology ePrint Archive*, 2015:309, 2015.

[74] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In Johansson and Nguyen [104], pages 537–556.

[75] Keith B. Frikken and Philippe Golle. Private social network analysis: how to assemble pieces of a graph privately. In Ari Juels and Marianne Winslett, editors, *Proceedings of the 2006 ACM Workshop on Privacy in the Electronic Society, WPES 2006, Alexandria, VA, USA, October 30, 2006*, pages 89–98. ACM, 2006.

[76] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 225–255, 2017.

[77] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 210–229. IEEE Computer Society, 2015.

[78] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, volume 9816 of *Lecture Notes in Computer Science*, pages 563–592. Springer, 2016.

[79] Adria Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies*, 4:248–267, 2017.

[80] Martin Joakim Bittel Geisler. *Cryptographic Protocols:: Theory and Implementation*. PhD thesis, Aarhus UniversitetAarhus University,[Enhedsstruktur før 1.7. 2011] Aarhus University, Det Naturvidenskabelige FakultetFaculty of Science, Datalogisk InstitutDepartment of Computer Science, 2010.

[81] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Johansson and Nguyen [104], pages 626–645.

[82] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew K. Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[83] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2014.

[84] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM J. Comput.*, 41(6):1673–1693, 2012.

[85] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 1069–1083. USENIX Association, 2016.

[86] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game - or - a completeness theorem for protocols with honest majority. In *Proc. 19th Symposium on Theory of Computing (STOC '87)*, pages 218–229, New York, 1987. ACM.

[87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194. ACM, 1987.

[88] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[89] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 95–100. ACM, 2011.

[90] Michael T. Goodrich and Roberto Tamassia. *Algorithm design - foundations, analysis and internet examples*. Wiley, 2002.

[91] S. Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. Complete fairness in secure two-party computation. *J. ACM*, 58(6):24:1–24:37, 2011.

[92] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. Mpc-friendly symmetric key primitives. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 430–443. ACM, 2016.

[93] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[94] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. *IACR Cryptology ePrint Archive*, 2015:751, 2015.

[95] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.

[96] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT*, 2017.

[97] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 150–169. Springer, 2015.

[98] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[99] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[100] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* [1].

[101] Nachikethas A Jagadeesan, Ranjan Pal, Kaushik Nadikuditi, Yan Huang, Elaine Shi, and Minlan Yu. A secure computation framework for sdns. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 209–210. ACM, 2014.

[102] Roman Jagomägis. Secrec: a privacy-aware programming language with applications in data mining. *Master's thesis, University of Tartu*, 2010.

[103] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 955–966. ACM, 2013.

[104] Thomas Johansson and Phong Q. Nguyen, editors. *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*. Springer, 2013.

[105] Zach Jorgensen, Ting Yu, and Graham Cormode. Conservative or liberal? personalized differential privacy. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1023–1034. IEEE, 2015.

[106] Jonathan L. Dautrich Jr., Emil Stefanov, and Elaine Shi. Burst ORAM: minimizing ORAM response times for bursty access patterns. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 749–764. USENIX Association, 2014.

[107] Jaeyeon Jung and Thorsten Holz, editors. *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. USENIX Association, 2015.

[108] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Secure multi-party differential privacy. In *Advances in neural information processing systems*, pages 2008–2016, 2015.

[109] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

[110] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 506–525, 2014.

[111] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In *ASIACRYPT (2)*, pages 506–525, 2014.

[112] Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 549–560. ACM, 2013.

[113] Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 703–714. ACM, 2011.

[114] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *International Conference on Applied Cryptography and Network Security*, pages 566–584. Springer, 2014.

[115] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. *IACR Cryptology ePrint Archive*, 2017:799, 2017.

[116] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2014.

[117] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[118] Eyal Kolman and Benny Pinkas. Securely computing a ground speed model. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(4):54, 2017.

[119] Eyal Kushilevitz and Tal Malkin, editors. *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, volume 9563 of *Lecture Notes in Computer Science*. Springer, 2016.

[120] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 189–200. ACM, 2012.

[121] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 106–115, 2007.

[122] Yehuda Lindell. Fast Cut-and-Choose-Based Protocols for Malicious and Covert Adversaries. *Journal of Cryptology*, 29(2):456–490, 2016.

[123] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.

[124] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* [2], pages 359–376.

[125] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. *IACR Cryptology ePrint Archive*, 2017:452, 2017.

[126] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Johansson and Nguyen [104], pages 719–734.

[127] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. *L*-diversity: Privacy beyond *k*-anonymity. *TKDD*, 1(1):3, 2007.

[128] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[129] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[130] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 94–103. IEEE, 2007.

[131] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. Computational differential privacy. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2009.

[132] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 862–873. ACM, 2015.

[133] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, 1999.

[134] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 111–125. IEEE, 2008.

[135] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* [2], pages 377–394.

[136] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 377–394. IEEE, 2015.

[137] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A New Approach to Practical Active-Secure Two-Party Computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[138] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, 2009.

[139] Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *TCC*, volume 5444, pages 368–386. Springer, 2009.

[140] Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. Smooth sensitivity and sampling in private data analysis. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 75–84. ACM, 2007.

[141] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303. ACM, 1997.

[142] Pascal Paillier et al. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, volume 99, pages 223–238. Springer, 1999.

[143] Alisa Pankova. *Efficient Multiparty Computation Secure against Covert and Active Adversaries*. PhD thesis, Institute of Computer Science, University of Tartu, 2017.

[144] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. Dstress: Efficient differentially private computations on distributed data. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 560–574. ACM, 2017.

[145] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. Dstress: Efficient differentially private computations on distributed data. In *EuroSys*, pages 560–574, 2017.

[146] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252, 2013.

[147] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2008.

[148] Martin Pettai and Peeter Laud. Combining differential privacy and secure multiparty computation. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 421–430. ACM, 2015.

[149] Benny Pinkas. Recent advances in private set intersection (psi). http://www.cs.bris.ac.uk/Research/CryptographySecurity/TPMPC/Slides2017/BennyPinkas.pdf, 2017.

[150] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[151] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *IACR Cryptology ePrint Archive*, 2016:930, 2016.

[152] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 655–670. IEEE, 2014.

[153] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jung and Holz [107], pages 415–430.

[154] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 235–259, 2017.

[155] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, Technical report, SRI International, 1998.

[156] Thomas Schneider and Michael Zohner. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, volume 7859 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2013.

[157] Claus-Peter Schnorr. Efficient identification and signatures for smart cards (abstract). In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 688–689. Springer, 1989.

[158] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *International Conference on Applied Cryptography and Network Security*, pages 346–366. Springer, 2016.

[159] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[160] Abhi Shelat and Muthuramakrishnan Venkitasubramaniam. Secure computation from millionaire. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 736–757. Springer, 2015.

[161] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((log n)^3) worst-case cost. *IACR Cryptology ePrint Archive*, 2011:407, 2011.

[162] Sander Siim. A Comprehensive Protocol Suite for Secure Two-Party Computation. Master's thesis, Institute of Computer Science, University of Tartu, 2016.

[163] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* [1].

[164] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013.

[165] Madhu Sudan, editor. *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*. ACM, 2016.

[166] Latanya Sweeney. Simple demographics often identify people uniquely. *Health (San Francisco)*, 671:1–34, 2000.

[167] Lu Tian, Bargav Jayaraman, Quanquan Gu, and David Evans. Aggregating private sparse learning models using multi-party computation. In *NIPS Workshop on Private Multi-Party Machine Learning, Barcelona, Spain*, 2016.

[168] Tomas Toft. A secure priority queue; or: On secure datastructures from multiparty computation. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 20–33. Springer, 2013.

[169] Meilof Veeningen. Pinocchio-based adaptive zk-snarks and secure/correct adaptive function evaluation. In Marc Joye and Abderrahmane Nitaj, editors, *Progress in Cryptology - AFRICACRYPT 2017 - 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*, volume 10239 of *Lecture Notes in Computer Science*, pages 21–39, 2017.

[170] Nikolaj Volgushev, Andrei Lapets, and Azer Bestavros. Scather: Programming with multi-party computation and mapreduce. www.cs.bu.edu/techreports/pdf/2015-010-scather.pdf, 2015.

[171] Justin Wagner, Joseph N Paulson, Xiao Wang, Bobby Bhattacharjee, and Héctor Corrada Bravo. Privacy-preserving microbiome analysis using secure computation. *Bioinformatics*, 32(12):1873–1879, 2016.

[172] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[173] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-Scale Secure Multiparty Computation, 2017. Available from `http://eprint.iacr.org/2017/189`.

[174] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226. ACM, 2014.

[175] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[176] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[177] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

[178] A. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society, 1982.

[179] A. Yao. How to generate and exchange secrets. In *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS '86)*, pages 162–167. IEEE Computer Society, 1986.

[180] S. Zahur, X. Wang, M. Raykova, A. Gascon, J. Doerner, D. Evans, and J. Katz. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 218–234, May 2016.

[181] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 493–507, 2013.

[182] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *IACR Cryptology ePrint Archive*, 2015:1153, 2015.

[183] 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.

[184] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 218–234, 2016.