TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master Thesis

Secure Two-Party Computation: ABY versus Intel SGX

Susanne Felsen

January 10, 2019

Engineering Cryptographic Protocols
Department of Computer Science
Technische Universität Darmstadt

Supervisors: M.Sc. Christian Weinert
Prof. Dr.-Ing. Thomas Schneider

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Susanne Felsen, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Susanne Felsen, have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt,

Susanne Felsen

Abstract

Secure Two-Party Computation (2PC) allows two parties to jointly evaluate a publicly known function without disclosing their respective private inputs. This can be realized using well-known cryptographic protocols such as Yao's garbled circuit protocol or the protocol of Goldreich, Micali and Wigderson (GMW), both of which obliviously evaluate a Boolean circuit that represents the desired functionality. The ABY framework (Demmler et al., NDSS'15) provides state-of-the-art implementations of these protocols. Nonetheless, they incur high computational and communication overheads compared to the unprotected computation.

Our work explores an alternative approach to the problem of secure computation, which is based on *Intel Software Guard Extensions (Intel SGX)*. Intel SGX allows applications to create trusted execution environments called enclaves, which are isolated from all other software running on the same machine. In a two-party computation scenario, the two cooperating parties can both submit their private input to the enclave, inside which the function can then be computed securely and from which they can afterwards retrieve the result.

Unfortunately, Intel SGX does not provide protection against software side-channel attacks and researches have proven its vulnerability to them time and again. In contrast, Boolean circuits as used in cryptographic 2PC protocols are inherently secure against a number of side-channel attacks. To that effect, we propose evaluating a Boolean circuit representation of the function inside the enclave. We design an Intel SGX-based 2PC protocol and provide a proof-of-concept implementation putting it into practice. We benchmark our Intel SGX-based 2PC solution and compare it to an ABY-based solution. For the evaluation of a Boolean circuit with AND size 1 million and AND depth 1000, the communication of our implementation is four orders of magnitude lower compared to Yao's garbled circuit protocol and the GMW protocol. In a high-latency network setting, the run-time for our protocol's 2PC phase is reduced by factors 1.3× and 19×, respectively.

Acknowledgments

At this point, I would like to thank my thesis supervisors, Christian Weinert and Prof. Dr.-Ing. Thomas Schneider, for providing me with this interesting (but also sometimes frustrating) research topic and for the opportunity to work in their group. I am particularly grateful to Christian for his quick replies to my e-mails at any time of day or night and for all the helpful comments and feedback, I received.

I would also like to thank Ágnes Kiss for being so kind as to generate the universal circuits for me and for answering any questions I had on the topic.

Contents

1 Introduction

Secure Multi-Party Computation (MPC) allows a set of parties with private inputs to jointly compute a function of these inputs without revealing anything but the result. The focus of this work is on the two-party special case, which plays an important role in research and is called *Secure Two-Party Computation (2PC)* or *Secure Function Evaluation (SFE)*.
In the SFE setting, the function to be computed is publicly known. In cryptographic SFE protocols such as Yao's garbled circuit protocol [Yao86] and the protocol of *Goldreich, Micali and Wigderson (GMW)* [GMW87], it is typically represented as a Boolean circuit.
Whenever the function to be evaluated should also be kept secret, *Private Function Evaluation (PFE)* is used. With the help of universal circuits [Val76], the problem of private function evaluation can however be reduced to the problem of secure function evaluation. A universal circuit is a special type of Boolean circuit that can be programmed to simulate any Boolean function up to a given size. In the PFE setting, one party provides the private input and the other party provides the private function in form of the programming to the universal circuit. The universal circuit itself is public. The aforementioned cryptographic 2PC protocols can also be used to obliviously evaluate universal circuits.

The ABY framework [DSZ15] provides state-of-the-art implementations of Yao's garbled circuit protocol and the GMW protocol. It includes the latest 2PC improvements to maximize efficiency. Unfortunately, cryptographic two-party computation solutions based on these protocols still have a high computational and communication overhead compared to the unprotected computation. *Intel Software Guard Extensions (Intel SGX)* seems to be a promising alternative, which we explore in this work.
In an era of cloud computing, Intel SGX aims to solve the problem of *secure remote computation*. It allows applications to create trusted execution environments called enclaves, which are isolated from all other software running on the same machine, even privileged software. Enclaves can be used to protect security-sensitive code and data against modification and disclosure. In a two-party computation scenario, the two cooperating parties can both submit their private input to the enclave, inside which the function can then be computed securely and from which they can afterwards retrieve the result. For that purpose, both parties need to establish a trusted channel with the enclave. They can use *remote attestation* to verify the enclave's integrity before provisioning it with their secrets.

Due to the strong security guarantees it provides and its availability in commodity hardware, Intel SGX enjoys increasing popularity. Unfortunately, researchers have shown that Intel SGX is vulnerable to a variety of *software side-channel attacks* such as page fault attacks or cache attacks. These attacks can be used to compromise the confidentiality of SGX-protected data.

Contributions. Boolean circuits as used in cryptographic 2PC protocols are inherently secure against a number of side-channel attacks. In this work, we therefore propose to evaluate a Boolean function representation of the function to be computed inside the enclave, instead of evaluating the function in plain. We design an Intel SGX-based two-party computation protocol to that effect and provide a proof-of-concept implementation putting it into practice.
Due to the fact that we load the circuit into the enclave during the protocol execution, we make it possible to reuse the same enclave for the secure computation of many different functions. This makes our approach easy to use, even by non-experts. While many works have suggested using Intel SGX to realize privacy-preserving applications, most of them focus on a specific application scenario. In contrast, this work proposes a solution for general secure two-party computation, eliminating the need for a per-application redesign.
Moreover, to the best of our knowledge, we are the first to design and implement an Intel SGX-based solution to the problem of private function evaluation.

We benchmark our Intel SGX-based 2PC solution and compare it to an ABY-based cryptographic solution. To this end, we perform a differentiated analysis, measuring the run-times and communication required for evaluating circuits with different circuit structures, sizes and gate types in two different network settings. For the evaluation of a Boolean circuit with AND size 1 million and AND depth 1000, the communication of our Intel SGX-based approach is four orders of magnitude lower than that of Yao's garbled circuit protocol and the GMW protocol. In a high-latency network setting, the corresponding run-time is reduced by factors 1.3× and 19×, respectively.

Outline. Chapter 2 contains background information on secure function evaluation, private function evaluation and Intel SGX. For Intel SGX, we do not only explain the theoretical concepts but also describe how an Intel SGX application is practically implemented. We furthermore provide a detailed overview of Intel SGX's security with regards to software side-channel attacks. In Chapter 3, related works are summarized. In Chapter 4, we introduce our Intel SGX-based 2PC protocol and explain how we put it into practice in our proof-of-concept implementation. Chapter 5 contains the benchmarking results for our implementation and the ABY framework's implementations of Yao's protocol and the GMW protocol and a comparison of the three. Finally, we conclude our work in Chapter 6.

2 Background

This chapter aims to provide readers with background information on the topics that are relevant for the understanding of this work. Specifically, we discuss secure function evaluation in Section 2.1, private function evaluation in Section 2.2, the ABY framework Section 2.3 and Intel SGX in Section 2.4.

2.1 Secure Function Evaluation

Secure Function Evaluation (SFE) allows two mutually distrusting parties to jointly evaluate a publicly known function f on their respective private inputs x and y while ensuring that neither of them learns anything about the other party's input. In depends on the application, whether one or both of the parties receive the evaluation result $z = f(x, y)$. Typically, the values x, y and z are binary values with $x = (x_1, \ldots, x_{n_1})$, $y = (y_1, \ldots, y_{n_2})$ and $z = (z_1, \ldots, z_m)$, n_1, n_2 and m being the respective bit-lengths.

A classic problem which SFE can solve is the so-called millionaires' problem [Yao86] where two millionaires want to know who of them is richer without revealing their net worth to each other. SFE hereby constitutes an alternative to implementing a trusted third party.

The applications of SFE are manifold and continuously growing. They include auctions [NPS99], data mining [LP08], biometric identification [EFG+09; BCF+14], genome-wide association studies [TWSH18] and proximity testing [ŠG14; JKS+18] – whenever the goal is to perform them in a privacy-preserving manner.

Prominent examples of SFE protocols are Yao's garbled circuit protocol [Yao86] and the protocol of Goldreich, Micali and Wigderson (GMW) [GMW87], which are explained in more detail in Sections 2.1.2 and 2.1.3.

2.1.1 Building Blocks for SFE

Boolean Circuits. In many SFE protocols, the (Boolean) function to be computed is represented as a Boolean circuit. A Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ maps n binary inputs to m binary outputs. The corresponding Boolean circuit is a directed acyclic graph with n inputs, m outputs and l Boolean gates. The edges are called wires. An example of a Boolean circuit can be seen in Figure 2.1.

Both, Yao's garbled circuit protocol [Yao86] and the GMW protocol [GMW87] securely evaluate Boolean circuits. Since the evaluation of XOR gates is essentially "for free" in both protocols (i.e., requires only negligible computation and no communication, see Sections 2.1.2 and 2.1.3), the number of AND gates in a circuit is especially relevant. The *(multiplicative) size* of a circuit is defined as its number of AND gates, while the *(multiplicative) depth* of the circuit is defined as the maximum number of AND gates on any path from an input to an output of the circuit [SZ13].

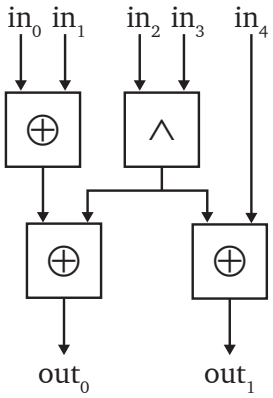


Figure 2.1: Boolean Circuit. Example Boolean circuit with $n = 5$ inputs, $m = 2$ outputs and $k = 4$ Boolean gates.

Oblivious Transfer. *Oblivious Transfer (OT)* [Rab81] is a two-party protocol executed between a sender and a receiver. In 1-out-of-2 OT, the sender inputs two values (x_1, x_2) and the receiver inputs a private selection bit σ. As a result of the protocol execution, the receiver learns x_σ but does not learn anything about $x_{1-\sigma}$ while the sender does not learn the receiver's selection bit σ.

OT is the foundation for almost all efficient SFE protocols. Yao's garbled circuit protocol [Yao86] requires one OT per input bit of one of the two parties while the GMW protocol [GMW87] requires two OTs per AND gate of the circuit for the pre-computation of multiplication triples during its set-up phase. The efficiency of the OT protocol is therefore extremely important. It can be greatly improved using OT pre-computation [Bea95] and OT extensions [IKNP03; ALSZ17].

Security Models. There are two widely used security models in the context of SFE. Many SFE protocols, including Yao's protocol [Yao86] and the GMW protocol [GMW87] in their basic form, only guarantee security in the weaker, *semi-honest* (or passive) model in which the protocol participants are assumed to be honest-but-curious. They follow the protocol specification but try to obtain as much information as possible from the protocol execution. This model allows for highly efficient protocol designs. Security in the *malicious* (or

active) model, on the other hand, is harder to achieve. In this model, the protocol participants may arbitrarily deviate from the protocol specification in order to obtain additional information.

2.1.2 Yao's Garbled Circuit Protocol

Yao's garbled circuit protocol [Yao86] was the first general two-party computation protocol. It was later proven secure by Lindell and Pinkas [LP09]. The protocol takes place between a *garbler* and an *evaluator*. The first step of the protocol is for the garbler to create a *garbled circuit* from the given Boolean circuit. He selects two random wire keys for each wire of the Boolean circuit, representing the values 0 and 1, respectively. He then creates a *garbled table* for each gate of the circuit. Obviously, the gate type defines which combination of input values results in which output value. The garbled table contains a random permutation of the corresponding output wire key encrypted with the two input wire keys for all four combinations of input wire keys. The garbler sends the garbled circuit along with his garbled private inputs to the evaluator. The evaluator has to obtain the garbled values corresponding to his own private inputs via 1-out-of-2 oblivious transfer. Consequently, Yao's garbled circuit protocol requires one OT per input bit of one of the two parties. In possession of all the input wire keys, the evaluator can now evaluate the garbled circuit gate-by-gate using the garbled tables until he reaches the output wires. If the evaluator is supposed to learn the result of the computation, the garbler can send him an *output decryption table* mapping the output wire keys to their corresponding plain text values. Otherwise, the evaluator sends the output wire keys back to the garbler.
The evaluation of the garbled circuit does not require any further interaction between the garbler and the evaluator. Therefore Yao's protocol has a constant number of rounds. As with oblivious transfer, there are extensions to the protocol which greatly improve its efficiency. Notably, an extension called *Free XOR* [KS08b] allows the "free" evaluation of XOR gates which is to say that the evaluation of a XOR gate does not require any communication and only negligible computation.

In its basic form, Yao's garbled circuit protocol provides security against semi-honest adversaries. It can however be extended for security against malicious adversaries [LP07].

2.1.3 The GMW Protocol

The protocol of *Goldreich, Micali and Wigderson (GMW)* [GMW87] also allows the secure evaluation of a Boolean circuit. It is based on *additive secret sharing*. Each input value v is shared among the two parties so that each of them holds a random-looking share v_i with $v = v_1 \oplus v_2$. Both parties then evaluate the circuit gate-by-gate. XOR gates can be evaluated locally since XOR is an associative operation. No interaction between the two parties is required. For an XOR gate with input values x and y and output value z, both parties compute their share of the output value as $z_i = x_i \oplus y_i$. The evaluation of AND gates does however require

interaction and can be achieved using *multiplication triples* [Bea92; SZ13]. Multiplication triples are random-looking shares $a_i, b_i, c_i, i \in \{0,1\}$ with $(c_1 \oplus c_2) = (a_1 \oplus a_2) \wedge (b_1 \oplus b_2)$. They are usually generated in the offline *set-up phase* which precedes the *online phase* in which the inputs are shared, the circuit is evaluated and finally, the output is reconstructed by adding up the shares of the output.
When evaluating an AND gate with input values x and y and output value z, both parties compute and exchange $d_i = x_i \oplus a_i$ and $e_i = y_i \oplus b_i$, after which they can compute $d = d_1 \oplus d_2$ and $e = e_1 \oplus e_2$. They can then compute their shares of the output value as $z_1 = (b_1 \wedge d) \oplus (a_1 \wedge e) \oplus c_1 \oplus (d \wedge e)$ and $z_2 = (b_2 \wedge d) \oplus (a_2 \wedge e) \oplus c_2$. This means that for every AND gate, two independent 2-bit messages have to be exchanged. In high-latency networks, this constitutes a performance bottleneck. Fortunately, all AND gates of the same circuit layer can be evaluated in parallel. The communication complexity of the GMW protocol therefore depends on the depth of the circuit and the best performance results can be achieved using depth-optimized circuit constructions. In settings with low network latency, the GMW protocol can even outperform Yao's garbled circuit protocol [SZ13].

In its basic form, the GMW protocol provides security against semi-honest adversaries. It can however be extended for security against malicious adversaries [NNOB12].

2.2 Private Function Evaluation

Private Function Evaluation (PFE) allows two mutually distrusting parties to jointly evaluate a private function f provided by one of the parties on a private input x provided by the other party while ensuring that neither of them learns anything about the other party's input. In the PFE setting, only the party which provided the private input receives the evaluation result $z = f(x, y)$.

PFE is used whenever the function to be evaluated should be kept secret. This is the case when proprietary software is supposed to be run on private data. In that line, researchers have proposed privacy-preserving solutions for credit checking [FAZ05], remote diagnostics [BPSW07], medical diagnostics [BFK+09] and intrusion detection [NSMS14]. Conceivable future applications of PFE include privacy-preserving billing protocols as might be used in the context of smart cars or smart homes when computing individual insurance or electricity rates.

One approach to PFE is based on universal circuits, background information on which can be found in the following paragraph.

Universal Circuits. A *Universal Circuit (UC)* is a special type of Boolean circuit, which can be programmed to simulate any Boolean function up to a given size k. In addition to the private input $x = (x_1, \ldots, x_n)$, a UC takes $p = (p_1, \ldots, p_q)$ private programming bits as input. The same UC can be used to compute many different functions by specifying

different programming bits. In other words, the concrete function f is given as the programming p_f to a universal circuit, such that it computes $z = f(x)$ for any input x. In short: $UC(x, p_f) = f(x)$.
Universal circuits were introduced by Valiant [Val76], who proposed two asymptotically size-optimal constructions with size $\Omega(k \log k)$ and depth $\Omega(k)$, where k is the size of the Boolean circuit representation of the simulated function f. Valiant's two constructions are based on 2-way and 4-way recursive structures, respectively.
The first implementation of UC-based PFE was provided by Kolesnikov and Schneider [KS08a]. Their UC construction had size $\Omega(k \log^2 k)$, i.e., an asymptotically non-optimal size. Later, Valiant's size-optimal 2-way UC construction was brought into practice by Kiss and Schneider [KS16] and by Lipmaa et al. [LMS16] in concurrent and independent works. Recently, Günther et al. [GKS17] provided an implementation of Valiant's 4-way UC and proposed an even more efficient hybrid UC construction.

Universal Circuit-Based PFE. Universal circuits allow reducing the problem of PFE to the problem of SFE. One party provides the private input x, the other party provides the private programming bits p_f for the UC. The UC can be made public. Due to the properties of SFE, nothing apart from the circuit size and the number of inputs and outputs is revealed.

It becomes apparent that UC-based PFE can easily be integrated into an SFE framework. PFE can be implemented using different underlying SFE protocols such as Yao's garbled circuit protocol [Yao86] or the GMW protocol [GMW87] (see Sections 2.1.2 and 2.1.3) and can therefore directly benefit from recent SFE protocol optimizations.

2.3 The ABY Framework

ABY[1] [DSZ15] is a framework for efficient mixed-protocol secure two-party computation. It combines secure computation schemes based on *arithmetic sharing*, *Boolean sharing* and *Yao sharing* which is also where its name comes from. The latter two sharings are used to evaluate functions represented as Boolean circuits, using the GMW protocol [GMW87] and Yao's garbled circuit protocol [Yao86], respectively. Explanations of these protocols can be found in Sections 2.1.2 and 2.1.3. UC-based PFE has also been integrated into ABY.
ABY is under continuous development and provides state-of-the-art implementations of all the protocols it supports, which includes the latest 2PC improvements to maximize efficiency. It allows developers to manually specify which part of a function should be computed in which sharing so that the every part can be computed in the sharing that is most efficient. ABY focuses on providing security in the semi-honest adversary model.

[1] The ABY source code is available online under `https://encrypto.de/code/ABY`.

The ABY Boolean Circuit Format. ABY defines the ABY Boolean circuit format[2]. It is human-readable and can be parsed by the ABY framework. The circuit format distinguishes between circuit input wires, circuit output wires and constant wires. In addition, there are function gates with one or more gate input wires and one gate output wire. Every wire has an individual wire ID and can serve as input to arbitrarily many function gates. The wires are sorted topologically meaning that a wire ID is always defined before it is used as a gate input or a circuit output. The ABY circuit format recognizes four different gate types: XOR, AND, MUX and Inversion. Figure 2.2 contains the ABY circuit format representation of a simple Boolean circuit.

```
1 # Circuit Input Wires
2 S 0 1 2 3 4
3 # Function Gates
4 X 0 1 6   # 6 = 0 XOR 1
5 A 2 3 7   # 7 = 2 AND 3
6 X 6 7 8   # 8 = 6 XOR 7
7 X 4 7 9   # 9 = 4 XOR 7
8 # Circuit Output Wires
9 O 8 9
```

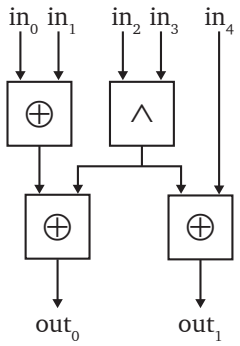


Figure 2.2: ABY Boolean Circuit Format. ABY Boolean circuit format representation (left) of a simple Boolean circuit (right).

The Universal Circuit Format. Kiss and Schneider [KS16] define a similar format for universal circuits[3], which can be parsed by the ABY framework. It is generated by their UC compiler, together with the corresponding programming file.
Universal circuits are made up of X switches, Y switches and universal gates. The values computed for the gate output wires depend on one or more programming bits, which are read from the programming file. Every gate uses one line of the programming file. X switches and Y switches take a single programming bit. An X switch has two gate input and two gate output wires, either outputting them in order (if $p = 0$) or in reverse order (if $p = 1$). A Y switch also has two gate input wires but only one gate output wire, either outputting the value of the first input wire (if $p = 1$) or the second input wire (if $p = 0$). Universal gates take four programming bits and are able to compute any Boolean function with two inputs.Listing 2.1 shows how the different gate types are represented in the universal circuit format.

[2]A detailed description of the ABY Boolean circuit format can be found under `https://github.com/encryptogroup/ABY/blob/public/bin/circ/circuitformat.md`.
[3]A detailed description of the universal circuit format can be found under `https://github.com/encryptogroup/UC/blob/master/README.md`

Listing 2.1: Universal Circuit Format. Gate types and number of programming bits they require.

```
# Gate Types
X 0 1 2 3 #(p)
Y 4 5 6   #(p)
U 7 8 9   #(p1,p2,p3,p4)
```

2.4 Intel SGX

Intel Software Guard Extensions (Intel SGX) is an Intel architecture extension that adds support for *Trusted Execution Environments (TEEs)*. It consists of a set of new CPU instructions and memory access changes and was introduced in 2013 in three papers [MAB+13; HLP+13; AGJS13]. Intel SGX is a feature of all Intel Core processors from the 6th generation onwards and therefore ubiquitously available. It aims to be an application security solution for developers which is not dependent upon platform security.

With Intel SGX, an application is partitioned into an untrusted part and a trusted part called *enclave*. All security-critical parts of the application are moved into the enclave.
An enclave is a hardware-based application-layer TEE. It is instantiated by untrusted code. Its initial content (code and data) is loaded from unprotected memory and is free for inspection and analysis. It might even be subject to manipulation. Once the enclave has been initialized, its content is protected from modification and disclosure. The enclave memory cannot be accessed by the untrusted part of the application or other applications and it cannot even be accessed by privileged software. Confidential data can now be provisioned to the enclave using a secure channel.
Intel SGX includes a mechanism called *sealing*, which allows to persistently store the provisioned secrets or other confidential enclave data for future enclave executions.

In an era of cloud computing, Intel SGX aims to solve the problem of *secure remote computation*. It allows a *service provider* to securely execute software on a remote platform such as a cloud server that is owned by an untrusted party. Before rendering the service, i.e., before provisioning secrets to the enclave, the service provider can use an advanced feature of Intel SGX called *Remote Attestation (RA)* to verify the enclave's integrity and authenticity. This feature is explained in detail in Section 2.4.3. Prior to that, Section 2.4.1 provides an overview of the fundamentals of Intel SGX and Section 2.4.2 explains how an Intel SGX application is created in practice. Lastly, applications of Intel SGX are listed in Section 2.4.4 and the topic of side-channel attacks on Intel SGX is addressed in Section 2.4.5.
Additional information on Intel SGX and an in-depth explanation of the theoretical background can be found in [CD16].

2.4.1 SGX Fundamentals

Security Guarantees. For enclaves, Intel SGX guarantees *confidentiality* of data and *integrity* of execution. Software running inside an enclave is isolated from all other software running on the same machine. Enclave memory cannot be accessed from outside the enclave, not even by privileged system software such as the Operating System (OS), the Virtual Machine Manager (VMM), device drivers or the system BIOS. The *Trusted Computing Base (TCB)* excludes the system software and only includes the CPU (hardware and firmware) and the software inside the enclave. Intel SGX therefore offers protection against *privileged software attacks*. It aims at reducing the TCB to a minimum in order to expose an equally minimal attack surface.
Intel SGX does not protect against *software side-channel attacks* such as cache attacks or page fault attacks. Responsibility for addressing side-channel attacks is deferred to the developer [AGJS13; Int18e; Int18f].

Physical Memory Organization. The enclaves' code and data as well as associated SGX control structures are stored in the *Enclave Page Cache (EPC)* which is made up of 4 KB pages. Every EPC page has an associated *Enclave Page Cache Map (EPCM)* entry which contains security and access control information. The EPCM's contents are used by the CPU when performing memory access checks for the enclave pages during the address translation process. Every enclave has an associated *SGX Enclave Control Structure (SECS)* which contains per-enclave metadata and is stored in its own EPC page. The EPC and EPCM are stored inside a special range of the main memory (DRAM) called *Processor Reserved Memory (PRM)*. The PRM cannot be accessed by the system software or by peripherals. Its contents are encrypted and integrity protected by the *Memory Encryption Engine (MEE)*.
The system software can oversubscribe the EPC by securely evicting EPC pages to non-PRM DRAM. From there, they can be further evicted to disk by classical page swapping mechanisms. When an application tries to access a page that has been evicted, it is reloaded into the EPC. As this might in turn lead to the eviction of another EPC page, this process is called *EPC page swapping* or *EPC paging* [MAB+13; CD16].

Enclave Measurement & Signature. When an enclave is created and its pages are loaded into the EPC, their memory contents are cryptographically measured. The resulting *enclave measurement* not only identifies the contents of the enclave pages (initial code and data), but also their order and position as well as their security properties. Since the enclave measurement uniquely identifies an enclave, it is also called *enclave identity*. It is stored in the MRENCLAVE register.
In addition, an enclave also has a so-called *sealing identity*, which includes the *sealing authority*, *product ID (ISVPRODID)* and *security version number (ISVSVN)* of the enclave. The sealing authority is represented by the hash of the enclave author's public key. It uniquely identifies the enclave author.
The enclave author signs the enclave using an RSA key prior to distribution. More precisely,

it signs a certificate which also includes the expected values for the aforementioned enclave properties. The signing process produces the *Enclave Signature Structure (SIGSTRUCT)*. When the enclave is loaded, the hardware is presented with the SIGSTRUCT. It verifies the sealing authority's signature and checks that the value of the MRENCLAVE register matches the measurement specified inside the SIGSTRUCT. This way, modifications to the enclave file can be detected. If the checks pass, a hash of the sealing authority's public key is stored in the MRSIGNER register. Both, the enclave identity and the sealing identity are stored in the enclave's SECS [AGJS13; MAB+13; Int18e].

2.4.2 Programming with Intel SGX

Requirements. On any platform, applications can only use Intel SGX, if all of the following requirements are met [Int18g]:

- The CPU supports the Intel SGX instructions
- The system BIOS supports Intel SGX
- Intel SGX is enabled in the system BIOS
- The *Intel SGX Platform Software (PSW)* is installed

The Intel SGX PSW package includes the Intel SGX runtime system library, the *Application Enclave Service Manager (AESM)* and Intel SGX application enclaves, which are Intel-provided architectural enclaves such as the *Quoting Enclave* (see Section 2.4.3).

For developers, who wish to develop an Intel SGX application, Intel provides the *Intel SGX Software Development Kit (SDK)*. It contains trusted libraries, development tools and sample projects. The *Intel SGX Developer Guide* [Int18e], the *Intel SGX SDK Developer Reference* [Int18f] and the Intel website[4] render further assistance to developers.

Intel SGX applications are typically written in C/C++ but the *Rust SGX SDK*[5] [DDL+17] helps developers to write them in the Rust programming language. It is built upon the Intel SGX SDK and provides additional sample projects and third party libraries, which have been adapted for use within enclaves.

Development Process. A key aspect of developing an Intel SGX application is partitioning the application into a trusted and an untrusted part while keeping the TCB size in mind. An interface between the two parts has to be defined. It contains *Enclave Calls (ECALLs)*, which allow the untrusted part to call into the enclave, as well as *Outside Calls (OCALLs)*, which allow the enclave to call out to the outside application, e.g., to use operating system capabilities such as system calls. The interface is defined within an *Enclave Definition Language (EDL)* file. That file is used by the *Edger8r Tool* to generate proxy functions which can be invoked

[4] `https://software.intel.com/en-us/sgx`
[5] The Rust SGX SDK is available under `https://github.com/baidu/rust-sgx-sdk`.

by the application as any other function.
Given one EDL file, for example named `enclave.edl`, the Edger8r Tool generates four files: `enclave_t.h`, `enclave_t.c`, `enclave_u.h` and `enclave_u.c`. For every ECALL and OCALL, a trusted and an untrusted proxy function is created. The first two files contain the declaration and definition of the trusted proxy function while the last two files contain those of the untrusted proxy function. When making an ECALL, the untrusted part of the application will first call the untrusted proxy function, which in turn calls the trusted proxy function inside the enclave. The trusted proxy function finally calls the actual enclave function. When an OCALL is made, the sequence is reversed. The proxy functions act as wrappers and are responsible for bounds checking and marshalling data and pointers into and out of the enclave. Function parameters and buffers referenced by pointers are copied from protected enclave memory to unprotected memory and vice versa. The Intel SGX SDK abstracts from these low-level details and provides developers with a familiar programming environment [Int18e; Int16].
An enclave can be delivered as a shared library, which is loaded into protected memory (EPC) when the enclave is created. As already mentioned in Section 2.4.1, the enclave is signed by the enclave author. The Intel SGX SDK provides the *Enclave Signing Tool* (`sgx_sign`) for that purpose. It typically runs automatically at the end of the build process and generates the *enclave signature* (SIGSTRUCT), which is used to confirm that the enclave has not been tampered with and has been correctly loaded. To this end, the enclave signature includes the enclave measurement and identifies the enclave author. The Enclave Signing Tool takes the *Enclave Configuration File* as input. It sets the enclave properties inside the SIGSTRUCT according to the values specified by the developer in this XML file. These properties include the enclave product ID and security version number. Additionally, the developer can also specify the enclave's maximum stack and heap size and whether the enclave can be debugged within the enclave configuration file [MAB+13; Int18e; Int18f].

Intel SGX SDK Compilation Profiles. There are two modes of operations for enclaves: *debug mode* and *production mode*. In contrary to production mode enclaves, debug mode enclaves do not have the full protection provided by the Intel SGX architecture, so that development tools such as debuggers and performance analyzers have access.
Enclave developers can choose between four different compilation profiles:

- *Debug*: In this mode, compiler optimizations are disabled and symbol information is saved. Enclaves are launched in debug mode.
- *Release*: In this mode, compiler optimizations are enabled and no symbol information is saved. Enclaves are launched in production mode.
- *Pre-Release*: This mode is similar to Release mode, except that enclaves are launched in debug mode.
- *Simulation*: In this mode, the application is linked with libraries that simulate the Intel SGX instructions, which allows the enclave to be run on platforms without Intel SGX. Enclaves are launched in debug mode.

An enclave built in Release mode can only be loaded, if the developer has applied for a commercial use licence[6] and has consequently been whitelisted by Intel. The private signing key that is used to sign enclaves built in Release mode must be kept in a protected environment such as a hardware security module. Under these circumstances, a two-step signing process has to be performed [Int18c; Int18i].

The Alternative: Shielding Systems. The development effort for Intel SGX applications is clearly higher than for conventional applications. One reason for this is that Intel SGX imposes several restrictions on the enclave code. The Intel SGX SDK contains a trusted version of the C++ standard library, which does not support I/O-related functions or system calls [Int18f]. While the security guarantees offered by Intel SGX are no doubt desirable, its limitations and the imposed performance overhead are hindering its adoption. This is especially true when aiming to protect feature-rich, real-world applications or legacy applications, which might have complex OS dependencies. The protection of the latter requires significant code changes. To alleviate this problem, researchers have proposed different Intel SGX-based *shielding systems* [BPH14; TPV17; ATG+16; SLTS17; OTK+18] over the last few years. Shielding systems shield applications from the untrusted OS, e.g., validating all inputs received from it. By adding system and/or trusted library support to enclaves, they make it easier to secure applications with Intel SGX. The in-enclave support for system and/or library calls typically reduces the number of expensive thread transitions in and out of the enclave. At the same time, it also reduces the size and complexity of the interface between the enclave and the untrusted OS, which constitutes an attack surface for Iago attacks [CS13] from the OS. Pulling more functionality into the enclave does however also result in a significant increase of the TCB size. Since any vulnerability in the TCB could possibly result in a breach of enclave security, this has raised many concerns. Additionally, it is important to note that page swapping will however become necessary if the application and shielding system's memory requirements exceed the EPC size.
The shielding systems Haven [BPH14] and Graphene-SGX [TPV17] allow developers to run unmodified applications inside SGX enclaves by placing a library OS inside of the enclave. To achieve maximum compatibility, Graphene-SGX includes glibc[7], an implementation if the C standard library, inside the enclave, which on its own has more than a million lines of code. The authors do however note that unused code could be removed and the much smaller musl libc[8] implementation could be used for applications that do not need the additional features only provided by glibc. Aiming for a small TCB but also for a good compatibility with existing code, SCONE [ATG+16] includes the musl libc implementation inside the enclave. Panoply [SLTS17], in contrast, does not include any libc implementation, instead delegating all the system calls to the untrusted OS. The authors prioritize a minimal TCB over performance. While Panoply's performance is only slightly worse than that of Graphene-SGX and SCONE, it cannot run unmodified applications, therefore requiring greater porting

[6] Companies can apply for a commercial use licence under `https://software.intel.com/en-us/sgx/commercial-use-license-request`
[7] `https://www.gnu.org/software/libc/`
[8] `https://www.musl-libc.org/`

efforts. It is explicitly designed for multi-process applications. In contrast to SCONE, Panoply synchronously exits the enclave to perform system calls, resulting in higher enclave transition costs. SCONE, which is short for “secure container environment”, uses enclaves to isolate and protect docker containers from each other as well as from the privileged system software. While Graphene-SGX and Panoply are open-source, SCONE is not.
None of the aforementioned shielding systems target side-channel attacks. A new shielding system called Varys [OTK+18], whose implementation is based on SCONE, has however recently been proposed, offering protection against the majority of page table- and cache-based side-channel attacks. More information on these attacks and on how Varys attempts to prevent them can be found in Section 2.4.5.

2.4.3 Remote Attestation

Remote Attestation (RA) allows a *Service Provider (SP)* to verify the integrity of the code running inside an enclave on a remote SGX-enabled platform before rendering a service to it. A possible service could be the *provisioning of secrets*, before which the service provider wants to be sure that the enclave can be trusted and the secrets are well-protected. To ensure the secure delivery of the service, remote attestation is typically used during the establishment of a trusted channel.

Remote attestation relies on the ability of the SGX-enabled platform to produce a credential that accurately reflects the enclave and platform state. This credential is called a *quote*. It is generated by the *Quoting Enclave*, which is an Intel-provided architectural enclave devoted to remote attestation. The quote is signed using the *Intel Enhanced Privacy ID (Intel EPID)* signature scheme, which is explained below. Afterwards, the consecutive steps of the attestation process are detailed. Last, it is described how remote attestation can be implemented in practice.

Intel Enhanced Privacy ID. The *Intel Enhanced Privacy ID (Intel EPID)* signature scheme is used by the Quoting Enclave to sign quotes. The resulting signature can only be verified by the *Intel Attestation Service (IAS)*. As a group signature scheme, Intel EPID helps to preserve the signer’s privacy. In the case of Intel SGX, a signer is synonymous with an SGX-enabled platform. Each signer belongs to a group of signers. While every signer has their own private signing key, a single group public key can be used to verify the signatures produced by all group members. The EPID signature therefore does not uniquely identify the signer that produced it – it is anonymous. Not even the group issuer can determine which key was used to create it.
Intel EPID has two signature modes, producing *linkable* and *unlinkable* signatures, respectively. Developers can choose which signature mode they want to use. With linkable signatures, the verifier (Intel) is able to determine whether two signatures were produced by the same signer. For the verifier, this has the advantage that behaviour anomalies in the use of an EPID key can be detected and in case of compromise, all signatures generated by that EPID key

can be easily identified and revoked. Intel EPID does however include a general mechanism for revoking signatures and the corresponding signer's private key. This mechanism works on the basis of having the signers perform mathematical proofs to indicate that they did not create any of the signatures on the *Signature Revocation List (SigRL)* [JSR+16; Cha17; Int18e].

Remote Attestation Process. The remote attestation process consists of several steps which are explained in the following [AGJS13; Int18e].

1. When the enclave requires a service from outside the platform, it sends a provisioning request to the respective service provider. The service provider replies with a request for attestation. This request can have the form of a challenge, typically containing a nonce to guarantee liveliness.
2. Using an Intel SGX instruction, the enclave generates a report structure which contains enclave information such as the enclave measurement (MRENCLAVE), the enclave author (MRSIGNER), product ID (ISVPRODID), security version number (ISVSVN) and additional attributes (ATTRIBUTES), e.g., whether the enclave is running in debug mode (see Section 2.4.1 for details on these fields). The report further includes the security version number of the platform TCB (CPUSVN) and a data portion, which contains user data included by the enclave. The user data allows binding the secure channel that is being established to the RA process.
3. The report is delivered to the Quoting Enclave which verifies it.
4. The Quoting Enclave converts the report body into a quote which it signs using the EPID key. Only the Quoting Enclave has access to this key. The quote is then returned to the service provider.
5. The service provider uses the Intel Attestation Service (IAS) to verify the quote signature. The IAS also checks whether the TCB level of the platform is up-to-date. It returns a signed attestation report which contains the attestation status and additional information in case the attestation status was not "OK".
6. The service provider verifies the attestation report signature. It then compares the enclave and platform information inside the quote against a trusted configuration. If the two match, it renders the service to the application. Possible trust policies include only trusting an enclave with a specific measurement (MRENCLAVE) or a specific author (MRSIGNER) or requiring specific attributes to be set.

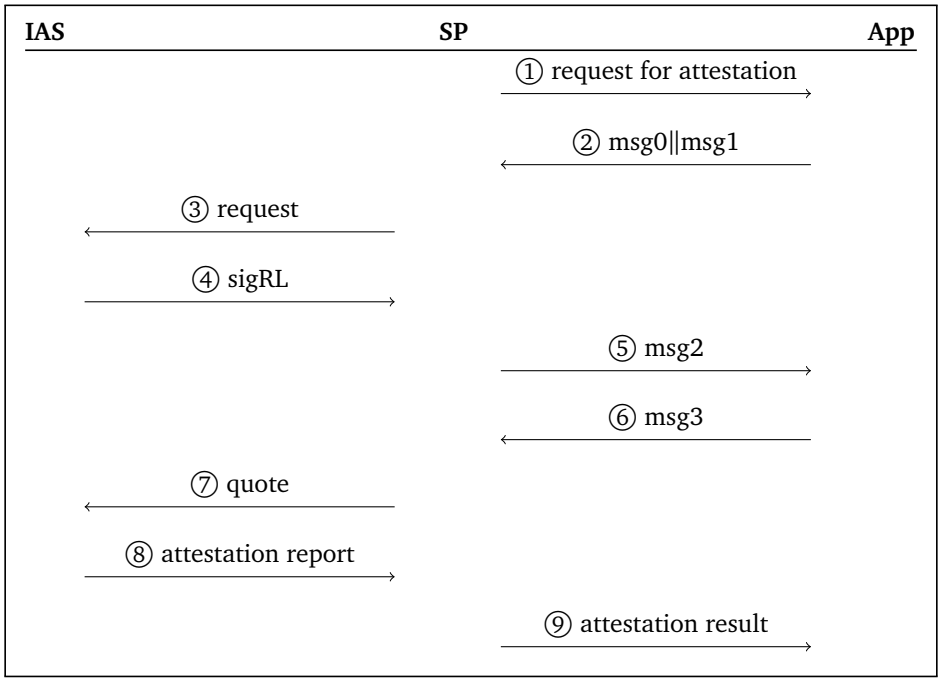
Remote Attestation in Practice. The following paragraph explains how remote attestation can be implemented in practice. As previously mentioned, remote attestation is typically used during the establishment of a trusted channel between the service provider and the enclave. The enclave proves that it was correctly instantiated and is running on a genuine SGX-enabled platform, trust is established and keys are exchanged. These keys can then be used for further, secure communication between service provider and enclave.

The Intel SGX SDK assists developers by providing remote *Key Exchange (KE)* libraries which implement a modified Sigma protocol which is used for *Diffie-Hellman Key Exchange (DHKE)*. They define protocol messages and wrapper functions for the entire attestation flow. SGX libraries typically come in pairs: a trusted library which is linked with the enclave and an untrusted library which is linked with the untrusted application. The corresponding EDL file also has to be imported.
In order to be able to use the Intel Attestation Service, the service provider first has to register with Intel[9]. Since the IAS uses *Mutual Transport Layer Security (MTLS)* for authentication, this requires sending in a valid X.509 client certificate. For testing purposes in a pre-production environment, this certificate is allowed to be self-signed. Furthermore, the service provider has to specify whether he wants to use linkable or unlinkable EPID signatures in enclave quotes. Intel then assigns a unique *Service Provider ID (SPID)* to the service provider, which is needed for communicating with the IAS.
The key exchange libraries require the service provider to have an ECDSA key pair for authentication. They use *Elliptic Curve Diffie-Hellman (ECDH)* for the actual key exchange. The service provider's public key is hardcoded inside the enclave. This is to make sure that the enclave can only communicate with the intended service provider. The established DH shared secret is used to derive a *Key Derivation Key (KDK)*, which in turn is used to derive multiple shared session keys for different purposes [Int18a; Int18b; Int18f].

The Rust SGX SDK includes a remote attestation code sample[10]. Intel also provides two: one[11] that is part of its SDK and one[12] that is not. The remote attestation process as implemented in practice in these code samples is detailed below. Protocol 2.1 shows the corresponding remote attestation message flow. The key exchange libraries defines the communication sequence for the messages that are exchanged between the service provider (SP) and the remote SGX application (App). The service provider requires the assistance of the Intel Attestation Service (IAS).

To begin, the service provider sends a request for attestation to the application (①). The application then executes an ECALL to a wrapper function that in turn executes `sgx_ra_init()` to initialize the RA process. The function takes the hardcoded service provider's public ECDSA key as an input and returns a DHKE context for later use. Additionally, the application calls `sgx_get_extended_epid_group_id()` to obtain the platform's extended Intel EPID group ID (GID). It can be sent to the service provider as message 0. Alternatively, message 0 can be sent together with message 1 to save one round trip (②). The service provider does however have to verify that the extended GID that it received is supported by the Intel Attestation Service. Currently, this is only true for an extended GID of zero. If a different value is received, the RA process has to be aborted.

[9] Developers can request access to the Intel SGX development services under `https://software.intel.com/en-us/form/sgx-onboarding`
[10] `https://github.com/baidu/rust-sgx-sdk/tree/master/samplecode/remoteattestation`
[11] `https://github.com/intel/linux-sgx/tree/master/SampleCode/RemoteAttestation`
[12] `https://github.com/intel/sgx-ra-sample`



Protocol 2.1: Remote Attestation Message Flow. Messages exchanged between the Service Provider (SP) and the Application (App) and between the Service Provider and the Intel Attestation Service (IAS) during the remote attestation process.

To obtain message 1, the application calls `sgx_ra_get_msg1()` which is part of the KE library. Message 1 contains the enclave's public key for DHKE, which is denoted by $Ga = (Ga_x, Ga_y)$, and the GID. Consequently:

$$msg0||msg1 = ExGID||Ga||GID$$

Upon receiving message 1, the service provider requests the *Signature Revocation List (SigRL)* for the remote platform's GID from the Intel Attestation Service (③, ④). The service provider then generates its own public key for DHKE denoted by $Gb = (Gb_x, Gb_y)$. It can now compute the DH shared secret from *Ga* and *Gb* and derive the *Key Derivation Key (KDK)* from it. With the help of a *Key Derivation Function (KDF)*, multiple shared keys serving different purposes are derived from the KDK. One of them is the *Session MAC Key (SMK)*.

To generate message 2 (⑤), the service provider further has to perform the following steps:

- Set the quote type which should be requested from the application, i.e., linkable or unlinkable. The service provider's SPID has to be associated with that quote type.
- Set the KDF_ID.

- Generate the ECDSA signature of $Gb_x||Gb_y||Ga_x||Ga_y$ using its ECDSA private key, producing $SigSP$.
- Compute the AES-128 CMAC of $A = Gb||SPID||Quote\_Type||KDF\_ID||SigSP$ using the SMK.

Message 2 is created as follows:

$$msg2 = A||CMAC_{SMK}(A)||SigRL, \quad \text{with } A = Gb||SPID||Quote\_Type||KDF\_ID||SigSP$$

The application processes message 2 using the library function `sgx_ra_proc_msg2()`. The function verifies the service provider's signature, checks the signature revocation list and returns message 3 (⑥), which contains the quote:

$$msg3 = M||CMAC_{SMK}(M), \quad \text{with } M = Ga||PS\_Security\_Property||Quote$$

When message 3 is received, the service provider first verifies that the value of Ga in message 3 matches the one in message 1. It then also verifies the CMAC.
As previously mentioned, the quote includes a data portion, containing user data included by the enclave. This way, the trusted channel that is being established can be bound to the RA process. The enclave is to include the SHA-256 digest of $Ga||Gb||VK$ in that data portion. The *Verification Key (VK)* is another one of the keys derived from the KDK. The service provider has to verify that the digest has been correctly included. If all of these checks pass, the service provider sends the quote to the Intel Attestation Service (⑦). The IAS verifies it and returns a signed attestation report (⑧). If the verification was successful and the SGX platform's TCB level was up-to-date, the attestation report contains the attestation status "OK". If this is not the case, it may contain attestation statuses such as "SIGNATURE_REVOKED", "KEY_REVOKED" or "GROUP_OUT_OF_DATE" in the event that the TCB level is outdated. In the latter case, the attestation report contains a so-called *Platform Information Blob (PIB)* which the service provider should forward to the application. The application can use the function `sgx_report_attestation_status()` to find out which of the TCB components on the SGX platform requires an update.
Upon receiving the attestation report, the service provider validates the attestation report signature. It then has to decide whether or not to trust the enclave based on its trust policy and the enclave and platform information contained in the quote. The attestation result (⑨) contains the service provider's trust decision. It may also contain the aforementioned PIB or secrets that the service provider is provisioning to the enclave, which are encrypted using a symmetric session key SK that is derived from the KDK. The enclave can obtain this key by calling the trusted library function `sgx_ra_get_keys()`. It can use it to decrypt the provisioned secrets and encrypt any computation results that should be sent back to the service provider.
Finally, when the secure channel is no longer needed, the session can be closed by entering the enclave and calling `sgx_ra_close()` [Int18a; Int18b; Int18f].

To conclude, it should be noted that the remote attestation code samples provided by the Intel SGX SDK and Rust SGX SDK include a sample cryptographic library (`libsample_`

`libcrypto.so`) that is used by the sample service provider and not meant for production use. It generates reproducible messages to aid in the debugging the remote attestation message flow. While the sample code may be very useful for getting started, it needs to be carefully inspected to figure out whether it is working as suspected. The RA code sample supplied by the Intel SGX SDK for example simulates the IAS. Smaller issues as e.g., found in the code sample from the Rust SGX SDK include not actually parsing the platform info blob but generating a default one.

2.4.4 Applications of Intel SGX

Due to the strong security guarantees provided by Intel SGX and its availability in commodity hardware, a wide variety of protocols and frameworks tailored to specific application scenarios have been proposed over the last few years. Shielding systems such as Haven [BPH14], Graphene-SGX [TPV17], SCONE [ATG+16], Panoply [SLTS17] and Varys [OTK+18] allow to run unmodified applications inside SGX enclaves (see also Section 2.4.2). Usage of Intel SGX is also frequently suggested for data processing and analytics in cloud computing environments. VC3 [SCF+15], one of the early applications of Intel SGX, allows to perform distributed MapReduce computations in an untrusted cloud. SecureKeeper extends Apache Zookeeper, which is a service for distributed coordination [BWG+16] and Opaque [ZDB+17] is an oblivious distributed data analytics platform. Microsoft Azure has rolled out SGX-capable servers [Rus17a], making the possibility of securing cloud applications with Intel SGX a reality.
Intel SGX is also often relied upon whenever data privacy is essential. There are Intel SGX-based approaches for secure genomic data analysis such as PREMIX [CDD+16] and PRINCESS [CWJ+17], privacy-preserving machine learning [OSF+16], privacy-preserving machine learning as a service [HSVW18], privacy-preserving speech processing [BFR+18] and private web search [MBF+17], just to name a few. Intel SGX can also be used for securing networking applications [KSH+15; PPFF16; KHH+17] or blockchain technologies such as smart contracts [KMS+16; LEPS16; ZCC+16; MHWK16; BCKS18]. Most notably, Intel SGX finds real-world use in the Hyperledger Sawtooth blockchain framework, where it secures the Proof of Elapsed Time (PoET) consensus protocol [Int] and Microsoft Azure's Confidential Consortium (Coco) Framework for blockchain networks [Rus17b]. It is also used for private contact discovery by the Signal messaging service [Mar18]. Even more works which focus on securing applications with Intel SGX are linked to on a page of the Intel SGX website dedicated to academic research[13].

2.4.5 Side-Channel Attacks on Intel SGX

Intel SGX does not provide protection against software side-channel attacks. The developers themselves are responsible for building enclaves that are protected against side-channel

[13] `https://software.intel.com/en-us/sgx/academic-research`

adversaries who may gather power statistics, cache miss statistics, branch statistics via timing or page access statistics via page tables [Int15]. This is extremely hard, especially since new side-channel vulnerabilities of Intel SGX are continuously being discovered by the research community.
A page on the Intel SGX website dedicated to academic research[14] lists some of the most relevant works on side-channel attacks targeting Intel SGX. Furthermore, Lindell [Lin18] gives a good general overview while Wang et al. [WCP+17] provide an in-depth analysis of memory side-channels.
Many of these attacks do not even require the attacker to have physical access to the machine running the victim enclave. If the implementation is not *constant-time*, meaning that its execution time depends on the secret data, that secret data can be deduced [BB03; BT11]. Furthermore, if different processes are executed on the same machine, secrets can be leaked due to the fact that the processes share physical resources such as for example caches. Therefore, achieving *co-location* of an attacker process on the same physical machine often suffices. In virtualized environments as used in cloud computing, multiple virtual machines (VMs) typically run on the same machine and researchers have shown that cache attacks can be performed across VMs running on different cores or even different CPUs of the same machine [OST06; RTSS09; ZJRR12; IES15]. Side-channel attacks such as cache attacks have also been shown to be practical in SGX environments. The different categories that side-channel attacks on Intel SGX can be divided into are briefly explained below.

Page Table-Based Attacks. In the SGX security model, the OS is untrusted. Nonetheless, it is relied upon for memory management, including paging, and more. A privileged attacker with control over the OS is able to manipulate the page tables, allowing him to induce page faults. By monitoring their occurrences, he can learn which pages were accessed. The resulting page-level access pattern was shown to be sufficient for extracting text documents or outlines of images from widely used application libraries [XCP15] and bits of encryption keys from cryptographic implementation libraries [SCNS16]. Later, Van Bulck et al. [VWK+17] as well as Wang et al. [WCP+17] were able to show that page accesses can also be inferred without inducing page faults, e.g., by monitoring page table attributes.
These page table-based side-channel attacks, which are also called *controlled-channel attacks*, exploit secret-dependent control transfers and data accesses. Mitigation strategies include placing sensitive data within the same page [SCNS16] or detecting the anomalously high exception or interrupt rate often associated with these attacks as T-SGX [SLKP17] and Déjà Vu [CZRZ17] do. These existing defenses fall short in the face of more sophisticated attacks that avoid producing too many interrupts [WCP+17]. An alternative approach is SGX-Shield [SLK+17], which implements fine-grained *Address Space Layout Randomization (ASLR)*. The memory layout is however only randomized at enclave load time and could still be learned by observing memory access patterns.

[14] `https://software.intel.com/en-us/sgx/academic-research`

Cache-Based Attacks. Memory caching is used by the CPU to reduce memory access times. A copy of the most recently accessed code and data is kept in cache memory, which is an order of magnitude faster and orders of magnitude smaller than the computer's main memory (DRAM). When a memory access is requested, the cache is checked for the requested data first. If it is found, a *cache hit* occurs and the request is served by reading from the cache. In contrast, in case of a *cache miss*, the data has to be retrieved from the next level of the memory hierarchy, from where it is copied to the cache. This typically leads to some other previously existing cache entry being evicted to make room. Cache fills and evictions operate on *cache lines*, which contain copies of contiguous ranges of DRAM and typically have a size of 64 bytes. Modern Intel CPUs have a three-level cache hierarchy. Each core has its own L1 and L2 cache while the L3 cache, which is also called *Last-Level Cache (LLC)*, is shared between all cores. The L1 cache is the smallest and fastest. In contrast to the L2 and L3 caches, it is divided into two separate caches for code and data [CD16].
Cache-based side-channel attacks take advantage of the fact that the time it takes to access a memory location depends on whether it has been cached or not and exploit secret-dependent memory accesses. Different cache attacks targeting different caches have been proposed. Popular attack techniques, which are often reused, include EVICT+TIME, PRIME+PROBE [OST06] and FLUSH+RELOAD [YF14]. Recent works [BMD+17; GESM17; HCP17; MIE17; SWG+17] have demonstrated that known cache attacks can also be performed on Intel SGX enclaves. They all use the PRIME+PROBE approach. First, the attacker process *primes* the cache, filling it with its own data. It then waits for the victim enclave to access the cache in a secret-dependent manner. This results in some of the attacker's cache lines being evicted from the cache. The attacker process then *probes* the cache by re-accessing the data that he previously loaded into the cache. From the measured access times the attacker can find out which of the cache lines were evicted. Due to the fact that a memory location is mapped to specific cache lines based on some of its address bits, this also reveals part of the memory address that was accessed by the victim. This has been shown to be enough to extract RSA private keys [BMD+17], AES keys [GESM17; MIE17] and sensitive information such as genomic data [BMD+17], images and parts of text documents [HCP17] from cryptographic and application libraries running inside enclaves. As the L1 cache was targeted, co-location on the same core was a prerequisite. An enclave-to-enclave cache attack targeting the LLC was demonstrated in [SWG+17].
Some, but not all, of the proposed attacks lead to an increased interrupt rate and can therefore be detected by existing defenses such as T-SGX [SLKP17] and Déjà Vu [CZRZ17]. These defenses do however also impose a noticeable overhead. T-SGX and Déjà Vu rely on *Intel Transactional Synchronization Extension (Intel TSX)*, which provides support for *Hardware Transactional Memory (HTM)*. HTM enables multiple threats to optimistically execute transactions in parallel, aborting and rolling back transactions in case of a conflict. Current HTM implementations use the caches to keep track of transactional changes. Transactions are therefore also aborted whenever transactional memory is prematurely evicted from the cache. Cloak [GLS+17a], another approach that aims at providing protection against cache-based side-channel attacks, leverages this behaviour of HTM implementations such as Intel TSX. It preloads all sensitive memory locations into the caches before accessing any of them in a possibly secret-dependent way. Cloak does however have the disadvantage of requiring

the developer to annotate sensitive data structures manually. Additionally, Intel TSX is not supported by all Intel SGX-enabled processors.
An alternative approach that has been proposed is DR.SGX [BCD+17], which is short for "Data Location Randomization for SGX". It prevents information leakage due to secret-dependent data accesses by continuously randomizing the enclave's memory layout and thereby obfuscating the link between memory locations and data objects.
Since cache attacks often target cryptographic libraries, great efforts have gone into designing side-channel resistant, constant-time variants of encryption algorithms [OST06; BGNS06]. The recently proposed MemJam attack [MES18] demonstrated the vulnerability of a constant-time AES implementation from the Intel IPP library, which is part of the Intel SGX SDK and was thought to be secure. The attack has a 4-byte intra-cache line granularity, breaking the assumption that constant cache line accesses prevent any leakage. Only code with true constant-time properties, including constant memory accesses can be expected to be leakage-free. The MemJam authors suggest exclusively using hardware-based or hardware-assisted implementations such as AES-NI. Unfortunately, hardware support for cryptographic primitives is limited and support for AES-NI can in some cases be disabled in the BIOS.

Speculative Execution-Based Attacks. Speculative execution is an optimization technique used by modern processors. The Meltdown [LSG+18] and Spectre [KHF+19] attacks have shown that it also opens the door for powerful side-channel attacks. Following their discovery, similar speculative execution-based attacks against Intel SGX enclaves were demonstrated. Processors speculatively execute instructions for example when reaching a conditional branch instruction, whose direction has yet to be determined. This might be because the direction depends on preceding instructions, whose execution is not yet completed or because the instruction is being executed *out-of-order* to further speed up the program. When this happens, the processor will make a prediction as to which path will be taken and continue executing the instructions along that path. If the prediction was correct, the execution results are committed. Otherwise, the instructions are rolled back. Their *transient execution* may however leave traces on the CPU's microarchitectural state such as the caches. Spectre attacks including SgxPectre [CCX+18] trick the processor into speculatively executing instruction sequences that are not part of the victim's intended execution path. Due to the transient execution of these instructions, information that the victim (e.g., enclave) is authorized to access is leaked. Similarly, Meltdown and Meltdown-type attacks such as Foreshadow [VMW+18; WVM+18; Int18h] exploit the fact that during a small time window the results of unauthorized memory accesses can be used in transient out-of-order instructions before they are rolled back. The attacker aims to transiently execute instructions that perform secret-dependent operations and alter the CPU's microarchitectural state, which is used as a *covert channel*, over which the secrets are transferred. Foreshadow for example uses the L1 cache as a covert channel. It is therefore also referred to as L1 Terminal Fault (L1TF). It comes in three variants, one of which targets Intel SGX enclaves, defeating memory isolation, sealing and attestation guarantees. Depending on a secret value, the location of a slot in an "oracle buffer" is computed. The slot is then brought into the cache, from where it can be recovered. This is done by measuring the time it takes to reload each slot of the oracle buffer.

Spectre-type attacks against Intel SGX require vulnerable code to be executed within the enclave. This is not the case for Meltdown-type attacks, which can even be performed without executing the victim enclave. The Foreshadow authors were able to extract enclave secrets residing in protected memory or CPU registers but more importantly, they were the first to be able to extract long-term keys from Intel-provided architectural enclaves such as the Quoting Enclave, thereby completely invalidating remote attestation guarantees. They showed that Meltdown-type attacks can also be used to breach non-hierarchical intra-address space isolation barriers. In contrast, the original Meltdown attack was used to breach the memory isolation barriers between kernel and user space, allowing an unprivileged user space attacker to read kernel memory. It has been mitigated using kernel page table isolation techniques [GLS+17b], which cannot defend against Foreshadow. Intel has however released a microcode update to protect enclaves from Foreshadow, which ensures that the L1 cache is flushed upon enclave exit. As secret data still resides in the L1 cache during enclave execution and the L1 cache is shared between logical cores, this leaves the possibility of cross-logical core attacks when HyperThreading is enabled. HyperThreading is Intel's proprietary implementation of *Simultaneous Multithreading (SMT)*. Intel has acknowledged the possible threat by deriving different keys depending on whether HyperThreading is enabled or disabled and including the status of HyperThreading in quotes. Consequently, service providers can decide for themselves whether or not to reject attestations from HyperThreading-enabled platforms. In the long run, in future Intel processors, Spectre, Meltdown and Foreshadow will reportedly be mitigated through hardware changes [Int18h; Int18j; WVM+18].

Bottom Line. The large and continuously increasing amount of side-channel attacks and corresponding mitigation strategies are hard to keep up with for any developer. It should be noted that this work does not aim to provide an exhaustive list. Many conventional side-channel attacks are amplified under the strong adversary model of Intel SGX and many existing mitigations require code changes or expert knowledge, or incur high performance costs. The underlying problem, as stated by Wang et al. [WCP+17], is that defenses are usually proposed in response to a newly discovered attack, only targeting this specific attack and failing to consider the bigger picture. Many defenses are therefore unable to offer protection against alternative attack strategies or even variations of the same attack.

Two recent works, HyperRace [CWC+18] and the shielding system Varys [OTK+18], do in fact aim to offer protection against a wider range of side-channel attacks. Both of them defend against interrupt-based as well as HyperThreading-based attacks. HyperThreading allows to run two concurrent threads on a single physical CPU core, improving processor performance. Consequently, each physical core is said to have two *logical cores*. Since the two logical cores share all the core resources such as the L1 and L2 cache, HyperThreading enables or assists same-core side-channel attacks, e.g., [WCP+17; GRBG18]. HyperThreading-based attacks typically do not trigger a large number of interrupts and are therefore not detected by many existing defenses. Nonetheless, due to the performance gains HyperThreading brings about, simply disabling it might not be in the developers' interest.
HyperRace and Varys ensure that the enclave thread is executed on a dedicated CPU core

that is not shared with untrusted threats. HyperRace creates a shadow thread, which it asks the OS to schedule on the physical core's second logical core. Since the OS is untrusted, HyperRace then verifies whether the two threads are indeed co-located on the same core. In principle, Varys does the same, except that it explicitly aims to protect multi-threaded applications, in that case scheduling two enclave threads on the same core.
Both, HyperRace and Varys additionally monitor interrupts. They therefore make page table-based side-channel attacks as well as L1/L2 cache attacks on enclaves much harder or even entirely impossible to mount. They are limited in that they do not protect against cross-core side-channel attacks such as LLC attacks. Cross-core side channels do however tend to be noisy and are often hard to exploit in practice.

On another note, assuming that developers can simply write code that is resistant to all sorts of side-channel attacks seems to be fairly unreasonable. The recently published MemJam attack [MES18] shows that even code that was written by experts and was thought to be secure can still be vulnerable. Additionally, some attacks like Foreshadow [VMW+18; WVM+18] do not require code vulnerabilities in the victim enclave.
Side-channel resistance is especially important for cryptographic implementations, which might otherwise leak secret keys. Hardware-assisted implementations such as AES-NI have so far withstood attacks, which is why their usage is strongly encouraged. In this line, it should be noted that while there is in-enclave support for AES-NI and the SGX Developer Guide [Int18e] references it as being resistant to timing-based side-channel attacks due to its constant-time properties, it has been repeatedly stated (e.g., in [GESM17; AM16]) that the Linux SGX SDK does not use it, instead using a slower software implementation without side-channel mitigations. While this is the default condition, Linux developers can manually link the SDK with a precompiled optimized binary of the Intel IPP library, which uses AES-NI, thereby improving the performance and security of their SGX applications [MES18; HTCK18]. Unfortunately, Intel itself does not provide any official information on this matter, making it hard to put into practice. On a brighter note, the Intel SGX SDK for Windows probably uses AES-NI by default.

3 Related Work

There are other works which investigate the use of trusted execution environments such as Intel SGX for secure computation. Koeberl et al. [KPR+15] were among the first to propose a TEE-based solution as a more efficient alternative to cryptographic secure multi-party computation protocols. While their work was purely theoretical in nature, they describe TEEs as neutral environments with strong protection, which enable multiple parties to jointly perform computations under previously agreed security and privacy policies. The authors accurately list assessing the security properties of a TEE-based solution, including its resistance to side-channel attacks, as one of the main challenges that have to be addressed.
Gupta et al. [GMF+16] suggest SGX-supported two-party secure function evaluation, providing a maliciously secure protocol that enables two parties with SGX-enabled machines to perform joint computations on their private data. They did not implement their proposed solution but expect it to be significantly more efficient than one based on garbled circuits due to the fact that much less cryptographic operations are required. The authors recognize the possibility of side-channel attacks such as memory side-channel attacks or timing attacks. The latter are protected against by including N extra loop iterations whenever the number of iterations is determined by a secret value, where N is a pseudo-random number which is based on secret information from both parties.
In this work, an Intel SGX-based two-party computation approach is practically implemented and compared against 2PC solutions based on cryptographic protocols. Additionally, an overview of Intel SGX's security with regards to software side-channel attacks is provided. Specifically, memory side-channel attacks and timing attacks are mitigated by evaluating a Boolean circuit-representation of the function to be computed inside the enclave and by performing constant memory accesses whenever a gate is evaluated.

Küçük et al. [KPM+16] explore the use of Intel SGX for many-party applications that involve thousands or tens of thousands of participants such as privacy-preserving energy metering or location-based services. Specifically, they use Intel SGX for implementing a *Trustworthy Remote Entity (TRE)*, which is a trusted third party providing strong assurance guarantees about its state and behaviour. The trustworthiness of this TRE is established via remote attestation. The authors imagine it to be an intermediary between data providers and data processors which can for example perform privacy-preserving operations. They implement a prototype TRE for the smart grid use case, which aggregates data from different energy meters. Due to the fact that all many-party applications share certain core features, they imagine it to serve as an architectural template for other applications. The authors furthermore emphasize the importance of minimizing the size of the TRE in order to minimize the TCB and the effort required to verify it. After benchmarking several SGX operations and

assessing the performance of their approach, they conclude that Intel SGX is well-suited for use in large-scale many-party applications, having a significant performance advantage over cryptographic MPC protocols.
This work obviously differs from Küçük et al.'s in that it only targets two-party applications. Extending it to the many-party case would however be feasible. What distinguishes this work from many others, including Küçük et al.'s, is that its design allows to reuse the same enclave for many different applications. This is due to the fact that the function to be computed is not executed in plain inside the enclave but instead, a Boolean circuit representation of it is executed. The circuit is loaded into the enclave during the protocol execution. By loading different circuits into the same enclave, it can be used to secure the computation of many different functions.
More works which focus on securing specific applications with Intel SGX are listed in Section 2.4.4. In contrast, this work proposes a solution for general secure two-party computation that does not require a per-application redesign.

Bahmani et al. [BBB+16; BBB+17] were the first to implement an Intel SGX-based approach for general secure multi-party computation. They provide a formal protocol specification and formal security definitions but lack to explain how they translate these into a practical implementation. At the center of their approach is the notion of labelled attested computation. It allows multiple parties, which concurrently and asynchronously interact with the same enclave, to obtain attestation guarantees. The parties establish individual secure channels with the remote program. The authors compare a two-party version of their Intel SGX-based solution against a cryptographic solution as implemented by the ABY framework. They do not specify what kind of network settings were used for the performance evaluation but report a speed-up of up to 300× for the total run time. It should however be noted that the performance gain for the online phase only is much smaller. In comparison to Intel SGX, the ABY framework provides weaker security in the semi-honest model. Bahmani et al. briefly address the topic of side-channel attacks against Intel SGX. They claim to provide a protocol implementation that is fully constant-time and thereby able to resist timing attacks.
Same as Bahmani et al.'s work, this work also implements an Intel SGX-based 2PC solution and compares its performance against an ABY-based solution but differs from it in that a Boolean circuit representation of the function to be computed is evaluated. The circuit is loaded into the enclave during the protocol execution. This does not only provide inherent security against a number of side-channel attacks but also makes it unnecessary to design a new enclave for every application. As a result the approach proposed in this work is much easier to use, even by non-experts. Additionally, it supports the secure evaluation of private functions. Further comparisons are hard without being provided with details about the implementation or even the protocol message flow.

4 Intel SGX-Based Secure Two-Party Computation

Secure two-party computation approaches based on cryptographic protocols incur high computational and communication overheads. This work explores an alternative Intel SGX-based approach to the problem of secure computation, which is described in this chapter. It targets secure function evaluation as well as private function evaluation. Both scenarios involve two mutually distrusting parties with private inputs that want to jointly evaluate a function while being sure that the other party does not learn their input.

Intel SGX allows applications to create trusted execution environments called enclaves which can be used to protect security-sensitive code and data against modification or disclosure. In a two-party computation scenario, the two cooperating parties can both submit their private input to the enclave, where the function can then be computed securely. Afterwards they can retrieve the computation result. For that purpose, the two parties need to establish individual trusted channels with the enclave. They can use remote attestation to verify the integrity of the enclave software before provisioning it with their secrets (see Section 2.4.3). This is the basic set-up of our 2PC approach.

Design Considerations. Intel SGX does not provide protection against software side-channel attacks and its vulnerability to them has been proven time and again (see Section 2.4.5). In our work, the function is not computed in plain inside the enclave. Instead, a Boolean circuit representation of it is executed, mitigating side-channel vulnerabilities. The main goal of this work therefore is to enable the execution of Boolean circuits and universal circuits within SGX enclaves.

Another goal of this work is to compare the performance of the Intel SGX-based solution against a cryptographic solution as implemented by the ABY framework. For this reason, circuits encoded in the ABY Boolean circuit format and the universal circuit format (see Section 2.3) are processed. The respective circuit is loaded into the enclave after the enclave's creation and can be selected per execution. This way, the same enclave can be reused for computing many different functions, which represents a big advantage. It however also means that the circuit is not covered by the enclave measurement, creating the need for a mechanism by which the two parties can be sure that the right circuit is being executed inside the enclave.

Basic Set-up. Figure 4.1 illustrates the basic set-up of our Intel SGX-based 2PC approach. The two cooperating parties are called service providers to match Intel's remote attestation terminology. The service they provide is the provisioning of secrets to the enclave. They both perform remote attestation and establish an individual trusted channel with the enclave, over which the secrets are exchanged. The following section describes the concrete protocol message flow between the service providers and the application. Inside the enclave, a Boolean circuit representation of the function to be computed is evaluated.

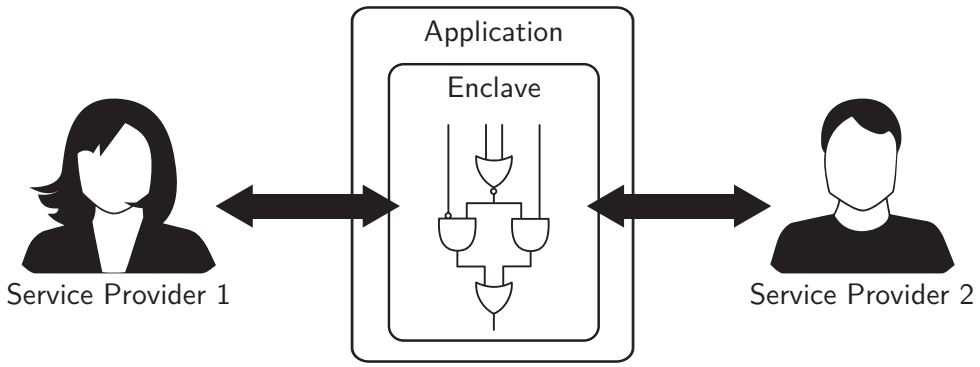
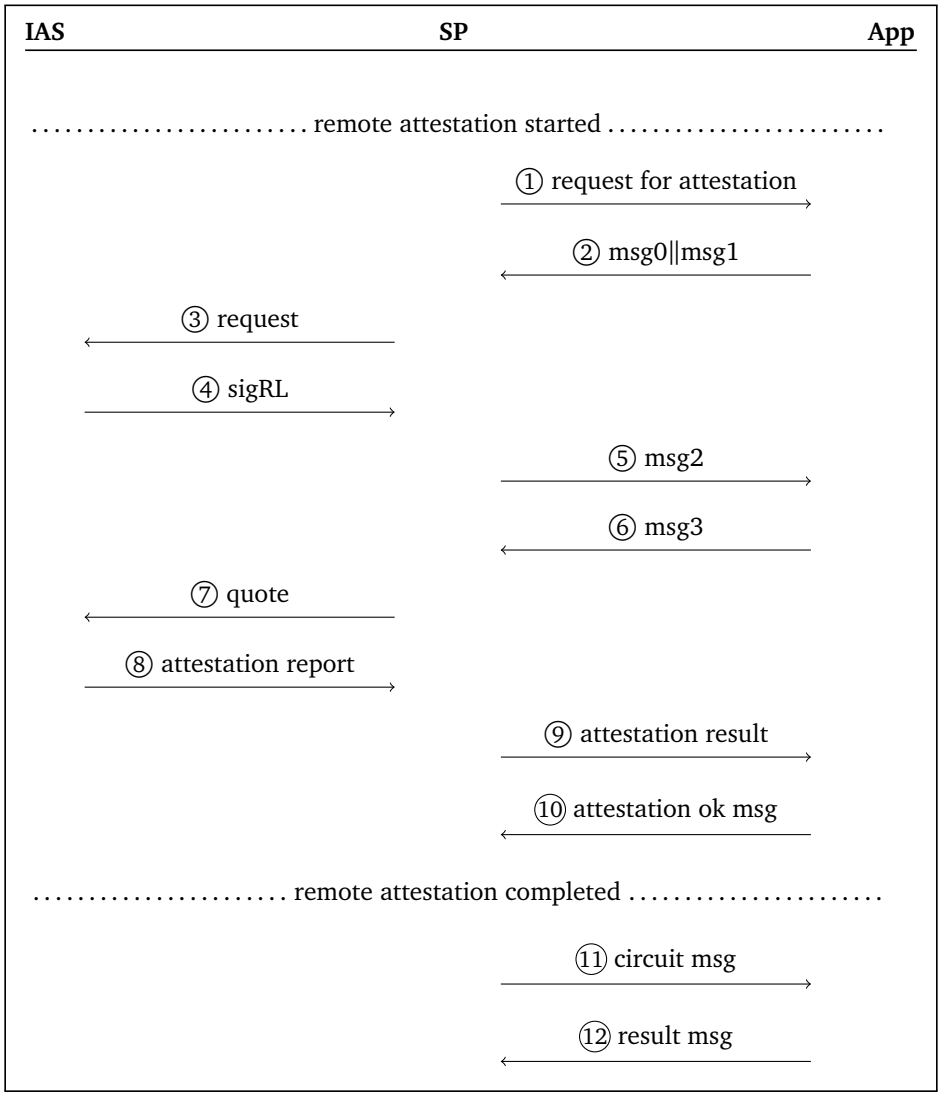


Figure 4.1: Basic Set-up. In the 2PC setting with Intel SGX, the two cooperating parties are called service providers because they provide the enclave with their secrets. Inside the enclave, a Boolean circuit is evaluated.

4.1 Protocol Message Flow

We designed a protocol for our Intel SGX-based 2PC approach. This section explains the protocol messages that are exchanged between each service provider and the application. The complete protocol message flow is shown in Protocol 4.1. It can be used for secure function evaluation as well as private function evaluation. Steps ① to ⑨ are part of the remote attestation process and are described in Section 2.4.3. If the RA process has been successfully completed and a trusted channel has been established between the service provider and the enclave, the application signals the service provider to provision its secrets over the newly established channel (⑩). We also count this step towards remote attestation.

In consequence, the service provider generates the circuit message (⑪). Despite its name, it does not actually contain the circuit to be evaluated in-enclave. Instead it contains a hash of it. Since Boolean circuits and especially universal circuits can get quite large for realistic applications, this significantly reduces the communication cost. The masking key MK and the symmetric key SK are two of the keys derived from the key derivation key. The MK is used to generate the AES-128 CMAC of the circuit hash in order to protect its integrity.



Protocol 4.1: Complete 2PC Protocol Message Flow for one Service Provider. Messages exchanged between each Service Provider (SP) and the Application (App) and between each Service Provider and the Intel Attestation Service (IAS) during the two-party computation which includes remote attestation.

The SK is used to encrypt the service provider's private input. As an encryption algorithm, AES in Galois/Counter Mode (AES-GCM) is used. AES-GCM produces a cipher text and an authentication tag, thereby providing both confidentiality and authenticity. Finally, the circuit message also contains the service provider's role in the 2PC computation. This role determines how the service provider's input is used, which enables using the same application for both secure function evaluation and private function evaluation. A wrong value will therefore lead

to the computation being aborted. In the PFE case, the private input of one of the parties is the programming to the universal circuit. This party does not receive the computation result. Summing it up, the circuit message has the following structure:

$$circuit\ msg = Role||CircuitHash||CMAC_{MK}(CircuitHash)||AES\text{-}GCM_{SK}(Input)$$

The application is responsible for loading the circuit into the enclave. This is only done once, after the first circuit message is received and requires identifying the circuit file corresponding to the circuit hash inside the message. The circuit file therefore needs to be stored on the remote platform. Since the application and the respective other service provider are untrusted, verifying whether the enclave contains the intended circuit before evaluating it is essential for security. The enclave code checks whether the hash of the actual circuit that was loaded into the enclave matches the circuit hashes submitted by both service providers. If this is not the case, the computation is aborted.

After the circuit inside the enclave has been initialized, the application performs an ECALL to pass the contents of the circuit message along to the enclave. It can then call into the enclave once more to check whether the computation result is available. If the enclave has received both service providers' inputs, this will trigger the evaluation of the circuit, given that the aforementioned checks have passed. If the second service provider's input has not yet been received, the first is informed accordingly and will have to request the result again. Otherwise, the computation result is encrypted with the respective shared secret key and returned to the service provider in form of the result message (⑫):

$$result\ msg = AES\text{-}GCM_{SK}(Result)$$

Upon receiving the message, the service provider decrypts the result. This concludes the secure two-party computation process.

4.2 Implementation

This section explains how our Intel SGX-based 2PC protocol presented in the preceding section is practically implemented.

The implementation was done using the Rust SGX SDK v1.0.4, which supports the Intel SGX SDK[1] v2.3.1. The development platform was Ubuntu 18.04 LTS.

[1]The Rust SGX SDK is available under `https://github.com/baidu/rust-sgx-sdk`.

4.2.1 Circuit Evaluation

This section briefly describes how Boolean circuits and universal circuits are represented and evaluated in this work.

A circuit is made up of nodes, which can be inputs, outputs or gates. The circuit files from which the circuits are parsed contain the nodes in topological order. To preserve this order, the circuit keeps a vector of node IDs. The node ID wraps the node type (input, output or gate) and an actual ID integer value. When the circuit is evaluated, the implementation iterates over the node IDs in the vector and looks up the corresponding node in one of three maps, each containing the nodes of one type. The circuit also keeps a lookup map with the circuit input and gate output values. Every gate has a gate type and a number of gate inputs and outputs. When a gate is processed, every input value is either a circuit input or the output of another gate that has already been processed and can therefore be looked up. The gate output value(s) are computed according to the gate type and inserted into the map. For side-channel protection, it is essential that the evaluation of a function gate is constant-time and that does not perform any secret-dependent cache accesses (see Section 2.4.5). Since the circuit topology is public and all possible paths of the circuit are always executed, the fact that a specific gate function is being evaluated does not leak any secrets. X switches, Y switches and universal gates are implemented using AND and XOR gates to avoid secret-dependent branching and data accesses.

The following two listings contain our implementations for the gate evaluation of Boolean circuits and universal circuits, respectively. The ABY Boolean circuit format supports four gate types, which are XOR, AND, MUX and Inversion. Universal circuits consist of X switches, Y switches and universal gates. The programming language is Rust. Our gate input and output values have the `u8` data type.

Listing 4.1: Boolean Circuit Gate Evaluation. Computation of the gate output value from the gate input value(s) for different gate types in a Boolean circuit.

```
#XOR
input_values[0] ^ input_values[1]

#AND
input_values[0] & input_values[1]

#MUX
input_values[0] & !(input_values[2] ^ 0b1111_1110)) |
    (input_values[1] & (input_values[2] ^ 0b1111_1110)

#Inversion
!(input_values[0] ^ 0b1111_1110)
```

Listing 4.2: Universal Circuit Gate Evaluation. Computation of the gate output value(s) from the gate input values for different gate types in a universal circuit.

```
#X Switch
let e = (input_values[0] ^ input_values[1]) & p;
e ^ input_values[0]
e ^ input_values[1]

#Y Switch
input_values[0] ^ input_values[1]) & p) ^ input_values[1]

#Universal Gate
let c = ((p1 ^ p2) & input_values[1]) ^ p1;
let d = ((p3 ^ p4) & input_values[1]) ^ p3;
(c ^ d) & input_values[0]) ^ c)
```

In order to achieve the best possible performance, we experimented with different data types for the four lookup maps that we use in our circuit implementation. As explained above, they contain the circuit input and output values, the gates and the gate output values, respectively. We started off using Rust's default hash map implementation (`HashMap`) and ended up using a map that is based on a B-tree (`BTreeMap`[2]). We also tried using `FxHashMap` from the fxhash library[3] and `FnvHashMap` from the fnv library[4] as adapted for use within enclaves by the Rust SGX SDK. In-enclave, using B-tree maps results in significantly shorter circuit evaluation times than can be achieved by using any of the hash map implementations. While Rust's `BTreeMap` reportedly provides excellent performance for small keys which are cheap to compare, its superior performance within enclaves is presumably mainly owed to the fact that it is more compact in memory, which leads to the circuit evaluation requiring less EPC paging.

4.2.2 2PC Protocol Implementation

In this section, we describe how we actually implemented the 2PC protocol we designed. Our implementation relies on the Rust SGX SDK's remote attestation code sample[5], which is a practical implementation of the remote attestation protocol as presented in Section 2.4.3. Further inspiration was taken from a code sample for private set intersection[6], which also extends the RA code sample.

[2] `https://doc.rust-lang.org/std/collections/struct.BTreeMap.html`
[3] `https://github.com/baidu/rust-sgx-sdk/tree/master/third_party/fxhash`
[4] `https://github.com/baidu/rust-sgx-sdk/tree/master/third_party/rust-fnv`
[5] `https://github.com/baidu/rust-sgx-sdk/tree/master/samplecode/remoteattestation`
[6] `https://github.com/baidu/rust-sgx-sdk/tree/master/samplecode/psi`

The Remote Attestation Code Sample. The RA sample project consists of two separate programs: the application, which includes the enclave, and the service provider. Only the former needs to be run on an SGX-enabled platform. The service provider and the untrusted part of the application are written in C++ while the enclave is written in the Rust programming language. Most notably, the RA code sample uses Google Protocol Buffers[7] for message exchange between the service provider and the application and Boost.Asio[8] for networking. By default, the communication between the service provider and the application is secured with TLS. The application takes the role of the TLS server, requiring a certificate for authentication.
Additional information on the RA code sample can be found in Appendix Section A.1.1. Before the code can be run, some set-up steps need to be performed which are explained in detail in Section 4.3.1.

Modifications & Extensions. We extended the RA code sample in a number of ways. Firstly, we added support for a second service provider, allowing the two service providers to both interact with the same enclave. Both service providers perform remote attestation and establish an individual trusted channel with the enclave. When the RA process is initialized, the service provider is assigned an RA context. This context is used to differentiate the two service providers and therefore included in all messages exchanged between the application and the service provider. In-enclave, it is used to retrieve the corresponding shared keys.
The application simultaneously communicates with both service providers. We aimed to implement the protocol in such a way that it is properly terminated in case of an error on any of the three sides.

The RA code sample was modified so that messages 0 and 1 are sent in conjunction instead of separately, eliminating one additional round-trip. For the 2PC computation, the following new message types were added:

- The *attestation failed message* that is sent by the application to the service provider in case the remote attestation process could not successfully be completed.
- The *circuit message* sent from the service provider to the application to transmit the circuit hash and the service provider's private input.
- The *result message* sent from the application to the service provider, containing the result of the two-party computation in encrypted form.
- The *result not ready message* that is sent by the application to the service provider in case the result could not yet be computed because the second service provider's input has not yet been received.
- The *result request message* sent by the service provider to the application, requesting the application to check whether the computation result is available. It will either be answered with a result message or a result not ready message.

[7] `https://developers.google.com/protocol-buffers/`
[8] `https://www.boost.org/doc/libs/1_68_0/doc/html/boost_asio.html`

- The *abort request message* sent from the service provider to the application in case it failed to retrieve the result after a certain number of tries. This will lead to the application checking whether the computation result is available one last time. If this is not the case, the computation is aborted and the enclave is re-initialized, clearing the secrets. This is to make sure that either both service providers receive the computation result or neither of them do.
- The *abort message* that is sent from the application to the service provider to signal that the computation has been aborted. This can be due to an error such as the circuit hash values received from the two service providers not matching or due to having received an abort request message from one of the service providers.

Generally, the actions of both, the application and the service provider, depend on the type of the messages that are received. We aimed to ensure that the computation is aborted and no information is leaked in case wrong values are included in messages or in case the messages are sent out of order. All security critical data such as the private input, result or circuit hash is confidentiality and/or integrity-protected.

In the RA code sample, the service provider uses a sample cryptographic library (`libsample_libcrypto.so`[9]). This leads to reproducible messages being generated and therefore facilitates the debugging of the remote attestation message flow. The library is not meant for production use which is why we replaced it. The replacement relies on OpenSSL's EVP interface[10] and is based on code from Intel's RA code sample[11] which was extended to support AES-GCM encryption and decryption.
Furthermore, smaller issues of the Rust SGX SDK's RA code sample were identified and fixed. In the RA code sample, the Platform Information Blob (PIB) that is transmitted by the Intel Attestation Service in case the attestation status is not "OK" is not actually parsed. Instead, a default one filled with zeros is generated and self-signed. The negative attestation status is also not reported on the application side, which is something that Intel recommends to do in order to find out which of the TCB components on the SGX platform requires an update. Additionally, the IAS signature on the attestation report is not verified. We resolved these issues, parsing the PIB, reporting the attestation status and verifying the IAS certificate chain and signature.

The RA code sample uses Google Protocol Buffers for message exchange. With quite some effort, we created a second version of our implementation without the additional TLS protection of the communication between the application and service provider to be able to inspect the plaintext messages exchanged between them using Wireshark[12]. A protocol buffer message is a series of key-value pairs, which are concatenated into a byte stream when the message is encoded[13]. The keys are the field numbers. For each field, a so-called *wire type* can

[9] `https://github.com/baidu/rust-sgx-sdk/blob/master/samplecode/remoteattestation/ServiceProvider/sample_libcrypto`
[10] `https://www.openssl.org/docs/man1.1.1/man7/evp.html`
[11] `https://github.com/intel/sgx-ra-sample/blob/master/crypto.c`
[12] `https://www.wireshark.org/`
[13] `https://developers.google.com/protocol-buffers/docs/encoding`

be chosen which determines how that field is encoded. It becomes obvious that using Protocol Buffers incurs some additional communication overhead. We tried to reduce this overhead by changing some of the wire types used in the RA code sample, yielding smaller message sizes, and picking the right wire types for our newly added message types. Nonetheless, in the PFE scenario, where one service provider's private input is the programming of the circuit, four bits are transmitted for each gate in the universal circuit due to the fact that universal gates require four programming bits. Since the circuit evaluation takes both the circuit input values and the programming bits as bytes (`u8` primitive type), the values need to be unpacked after they are decrypted inside the enclave.

To be able to measure the duration times of different phases of our protocol, we copied and adapted the timer implementation[14] provided by the ABY framework's utility library ENCRYPTO_utils.

Loading the Right Circuit into the Enclave. The central question during the implementation of the 2PC protocol was how to get the right Boolean or universal circuit into the enclave. This is actually a twofold problem: first, the circuit that is available in file form outside the enclave needs to be loaded into the enclave and second, the service providers need to be assured that the circuit they intend to be evaluated is being evaluated inside the enclave. Both, the case that the second service provider tries to get a different circuit evaluated and the case that the untrusted application or the untrusted OS loads the wrong circuit into the enclave need to be covered.
One of the first decisions we made was not to have the service provider transmit the circuit to the application during the protocol execution. The circuit is public, meaning that it does not have to be hidden and circuit sizes get quite large for realistic applications. To reduce the communication cost of the protocol, the circuit files are stored on the SGX-enabled platform. Here, we also want to point out that cryptographic 2PC solutions differ in whether they transmit the circuit or not. While in Yao's protocol, the garbled circuit is transferred from the garbler to the evaluator, the transfer of the circuit is not part of the GMW protocol.
Still, this leaves us with the problem of loading the right circuit into the enclave. As briefly mentioned, the second part of the problem is addressed by having the service providers transmit the hash value of the circuit they intend to be evaluated along with a CMAC of it to protect its integrity. When the circuit is loaded into the enclave, its hash value is computed. The circuit will only be evaluated if its hash value matches the values sent by the two service providers. For the first part of the problem, namely, how to get the circuit into the enclave, we considered several options. We first tried reading in and parsing the circuit file outside the enclave. The circuit object was then serialized and sent into the enclave via an ECALL, where it was deserialized and then hashed. This was somewhat challenging to implement. It required the help of the Rust SGX SDK's serialization library (`sgx_serialize`), which had to be extended to guarantee the deterministic serialization of the hash maps that were part of the circuit object by sorting their keys first. Without this, hashing the same circuit twice will result

[14] `https://github.com/encryptogroup/ENCRYPTO_utils/blob/master/src/ENCRYPTO_utils/timer.cpp`

in different hash values. The reading and the serialization of the circuit was implemented in Rust code which was called from C++ code via a Foreign Function Interface (FFI). As this approach did not yield the desired performance results, a second approach was implemented, which removes the serialization and deserialization steps. Instead of the serialized circuit object, the raw circuit file data is sent into the enclave, where it is hashed and parsed. This approach has a slightly better performance and also comes with a smaller TCB. To reduce the computation cost on the service provider side, the circuit hash is looked up rather than computed during the protocol execution.
One option we did not implement but which might still be worth exploring was the Intel Protected File System Library [Int18d] for secure file I/O inside enclaves. An enclave can use it to create SGX files, which are confidentiality and integrity protected, and to read them back in. It can however not be used to read in conventional files. For this purpose, the enclave would have to issue an OCALL. In theory, the circuit file could be read in and its contents could be copied into an SGX file during the protocol set-up phase, from where they could securely be read. This way, the expensive marshalling of the file data into the enclave would be avoided.

By loading the circuit into the enclave after its creation and by having the service providers select the circuit to be evaluated we are able to reuse the same enclave for many different applications.

4.3 Instructions for Use

This section contains instructions on how to use the Intel SGX-based 2PC solution that we designed and implemented. Before the code can be run as explained in Section 4.3.2, some set-up steps need to be performed. These are described in Section 4.3.1.

4.3.1 Set-up

Prerequisites. Our 2PC solution can either be run with our without Docker[15]. Docker allows to build and deploy applications with *containers*, which are isolated from each other. A Docker container is built from an image and packages the application and all of its dependencies such as tools and libraries. Docker significantly reduces the effort required for setting up the development environment and guarantees that the application always runs the same way by eliminating environment inconsistencies.
The publishers of the Rust SGX SDK recommend for it to be used with Docker. This only requires installing the Intel SGX Driver, cloning the GitHub repository and pulling the Docker image in order to start application development. Of course, Intel SGX has to be enabled in the system BIOS.

[15] `https://docs.docker.com/`

In order to be able to build and run an SGX application without Docker, the following components have to be installed[16]:

- Intel SGX Driver
- Intel SGX Platform Software (PSW)
- Intel SGX Software Development Kit (SDK)

Building and running our 2PC solution without Docker further requires the installation of the following external libraries:

- `libprotobuf-c0-dev`, `protobuf-compiler`
- `libboost-thread-dev`, `libboost-system-dev`
- `curl`, `libcurl4-openssl-dev`
- `libssl`
- `libjsoncpp-dev`
- `liblog4cpp5-dev`

Service Provider Registration. Access to the Intel SGX development services such as the Intel Attestation Service requires registering with Intel to obtain a valid Service Provider ID (SPID). The service provider has to send in a client certificate and chose an attestation policy. In this work, the unlinkable signature policy was chosen and the same SPID is used by both service providers. The service providers' ECDSA public key is hardcoded inside the enclave (`lib.rs`) to make sure that the enclave can only communicate with the intended service providers. The corresponding private key is hardcoded inside the `ServiceProvider.cpp` file. Both are in little endian format. If another SPID is to be used, these keys need to be replaced as well. If one wants to use the linkable signature policy, the quote type that is requested from the enclave has to be changed from "SGX_UNLINKABLE_SIGNATURE" to "SGX_LINKABLE_SIGNATURE". This can also be done inside the `ServiceProvider.cpp` file.

Configuration. The `GeneralSettings.h` file is used to configure both the application and the service provider. The following values have to be set:

- the application's IP address and port number
- the application's server certificate and corresponding private key for the TLS connection
- the service provider's ID (SPID) and client certificate including the corresponding private key
- the URL of the Attestation API (see also [Int18a])

[16] They can be downloaded under `https://01.org/intel-softwareguard-extensions/downloads` or `https://github.com/intel/linux-sgx`.

The file was extended to include the path to the signing certificate of the Intel Attestation Service, enabling the verification of the IAS signature on the attestation report.

As explained in Section 4.1, the circuit to be evaluated inside the enclave is not transmitted over the network. It needs to be stored on the SGX-enabled platform. The application furthermore needs a way to identify which circuit to load into the enclave. Since computing the circuit hash for all the available circuits until a match is found would be too time-consuming, the application is supplied with a file that maps circuit hash values to circuit file names. Analogously, the service provider is supplied with a file that maps circuit file names to circuit hashes. Both files can be auto-generated using a helper program (`abycircuitparser`) that iterates over all files in a directory. The settings file was therefore extended to include three additional values:

- the application's path to the directory containing the circuit files
- the application's path to the file that maps circuit hash values to circuit file names
- the service provider's path to the file that maps circuit file names to circuit hash values

The files have to contain one line per mapping and the key-value pair in that line has to be separated by a single space. The key-value pairs are read into a hash map during the initialization of the application.

4.3.2 Running the 2PC Protocol

This section provides information on how to run the 2PC protocol between the two service providers and the application on the SGX-enabled platform. Specifically, it explains which command-line parameters are required and how to start the necessary Docker containers.

Command-Line Parameters. The application does not take any command-line parameters but the service provider takes several, which are listed below and need to be specified in order. They are:

- the circuit file name
- the path to the service provider's private input
- the service provider's role in the 2PC computation

The circuit file name has to be contained in the file that maps circuit file names to circuit hash values. Both service providers have to specify the same circuit file name in order for the computation to be successful. The service provider's private input has to be supplied in form of a text file, which has to contain the input bit sequence in string form. The user can either specify an absolute path or a path relative to the `ServiceProvider` directory. The service provider's role in the 2PC computation is either client or server and is denoted by an integer.

In the SFE setting, one service provider's role should be 0 and the other one's role should be 1. The following lines show example commands which could be entered to run the two service providers:

```
./app circuit_file.aby resources/input1.txt 0
./app circuit_file.aby resources/input2.txt 1
```

In the PFE setting, the service providers' roles need to be 2 and 3. The input of the service provider which takes the role of the server (3) is used as the programming to the universal circuit and that service provider does not receive the result of the computation.

Logging Capabilities. Both, the application and the service provider use logging to make the individual steps of the SGX-based 2PC protocol observable and to report any errors to the user. Some more detailed information can be obtained by activating verbose logging. This will for example lead to the attestation report and circuit hash being printed and can be done inside the respective `isv_app.cpp` file by adding the following line of code:

```
LogBase::Inst()->Enable(log::verbose, true);
```

Working with Docker. For ease of use, we recommend running the application as well as both service providers each within their own Docker containers. For Ubuntu 18.04, this requires pulling the Docker image `baiduxlab/sgx-rust:1804`. The container for the application can then be started by entering the following commands:

```
docker run -p 127.0.0.1:22222:22222 -v /path/to/rust-sgx-sdk/:/root/sgx
-ti --device /dev/isgx baiduxlab/sgx-rust:1804
```

```
root@docker:/# /opt/intel/libsgx-enclave-common/aesm/aesm_service &
```

This has several effects. Port 22222 of the container is bound to port 22222 on 127.0.0.1 of the host machine. The port number and IP address should match the ones specified in `GeneralSettings.h` (see Section 4.3.1). A host directory is mounted as a volume in the container, which can access it via the path specified after the colon. In this example, the application code is located inside the `samplecode` subdirectory of the Rust SGX SDK. An interactive terminal session is started and the isgx kernel module is loaded. The container is based on the aforementioned image. Inside the container, the AESM service is started. Users can then navigate to the `Application` directory, where they can build and/or run the application.

The service providers do not require SGX support and can be run inside or outside a Docker container as desired, provided that the prerequisites listed above are fulfilled. When running them on the same platform as the application, it is the easiest to run them within containers created from the aforementioned image.

The containers need to be connected to the host network by entering:

```
docker run --net="host" -v /path/to/rust-sgx-sdk/:/root/sgx -ti
--device /dev/isgx baiduxlab/sgx-rust:1804
```

The computation result is written to files called `output_CLIENT.txt` and `output_SERVER.txt` within the service provider's `resource` directory.

5 Evaluation

In this chapter, we benchmark our Intel SGX-based solution for secure function evaluation and private function evaluation and compare it to a cryptographic solution as implemented by the ABY framework.

Section 5.1 contains information about our experimental set-up, including the circuits we used for benchmarking and the phases that we divided our protocol into. In Sections 5.2ff., we present our experimental results. They are discussed and compared in Section 5.5.

5.1 Experimental Set-up

5.1.1 Benchmarking Environment

General. We perform our benchmarks on an SGX-enabled Intel Compute Stick STK2mv64CC equipped with an Intel Core m5-6Y57 CPU with 1.10 GHz and 3,6 GB RAM, running Ubuntu 18.04 LTS.

We evaluate our solution using two realistic network settings, which are simulated using Linux Traffic Control: a low-latency *LAN setting* with a bandwidth of 1 GBit/s and a round-trip time of 1 ms and a high-latency *WAN setting* with a bandwidth of 100 MBit/s and a round-trip time of 100 ms.

Intel SGX. We manually built the Intel SGX PSW and the Intel SGX SDK v2.3.1 for Linux using a precompiled optimized binary of the Intel IPP library.
We use the newest Attestation API version of the Intel Attestation Service (IAS) [Int18a], which is version 3. This leads to the attestation status "CONFIGURATION_NEEDED" being returned along with a platform information blob. The status was added after the recent disclosures of HyperThreading-based attacks such as Foreshadow and indicates that HyperThreading is enabled on the SGX platform. Since the BIOS of the Intel Compute Stick that we used for our experiments does not include an option to disable HyperThreading, we were not able to switch it off. This leads to an additional 103 bytes being sent from each of the service providers to the application during the remote attestation process and two additional ECALLs being performed in order to report the respective attestation status to the enclave.

The enclave's maximum stack and heap size has to be specified within the enclave configuration file. The Enclave Page Cache (EPC) has a maximum size of 128 MB, out of which around 90 MB are usable. Whenever the sum of the stack and heap size exceeds 90 MB, EPC paging will occur. This has a negative impact on the performance because additional enclave transitions are required to handle page faults and because the pages that are swapped into and out of the EPC need to be decrypted and encrypted, respectively [WAK18].
The maximum stack and heap size required by the enclave can be determined with the help of the Enclave Debugger `sgx-gdb`, which is part of the Intel SGX SDK. To be able to evaluate circuits with 1 000 000 gates, we unfortunately had to set the stack size to 8 KB and the heap size to 205 MB.

The communication between the service providers and the application as well as between the service providers and the Intel Attestation Service is secured with TLS v1.2. The amount of data exchanged between the parties, including Ethernet, TCP/IP and TLS protocol overheads was determined with the help of Wireshark. Each Ethernet frame has at least 66 bytes. It includes a 14 byte Ethernet header, a 20 byte IP header and a 32 byte TCP header. The TLS-encrypted messages are sent over TCP and each TLS record comes with a 5 byte header. The messages exchanged between the service providers and the application are sent with a 20 byte header that includes the message length and type. This header is transmitted in its own TCP packet. The corresponding Ethernet frame has a total size of $66 + 49 = 115$ bytes.

5.1.2 Circuit Design

The performance of cryptographic 2PC protocols, namely the GMW protocol and Yao's garbled circuit protocol, is mainly determined by the circuit structure and the network latency. The evaluation of XOR gates is essentially "for free" in both protocols (i.e., requires only negligible computation and no communication, see Sections 2.1.2 and 2.1.3). In contrast, the evaluation of AND gates is costly. This is why the number of AND gates in a circuit, which is also called its AND size, is of special importance. The performance of the GMW protocol is additionally influenced by the AND depth of the circuit. Generally, during the online phase of the GMW protocol, two independent 2-bit messages have to be exchanged per AND gate. All AND gates of the same circuit layer can however be evaluated in parallel, which is why circuits with a small AND depth yield the best performance.
For a fair comparison of our Intel SGX-based SFE approach with cryptographic approaches based on the GMW protocol and Yao's garbled circuit protocol, we generate Boolean circuits with different circuit sizes, AND sizes and AND depths. Figure 5.1 shows the two basic structures of the generated circuits that are used for benchmarking. We call them *sequential* and *parallel* because in the first case, all gates of the circuit have to be evaluated sequentially and in the second case, all gates in the same layer can be evaluated in parallel. For each basic structure, we generate circuits with 100, 10 000 and 1 000 000 gates. We additionally generated sequential circuits with 10, 1 000 and 100 000 gates, which have the same AND depth as the just mentioned parallel circuits. For each circuit size, we construct circuits only consisting of AND gates, circuits only consisting of XOR gates and circuits consisting of alternating layers

of AND gates and XOR gates. All circuits have only two inputs. A sequential circuit has a single output while a parallel circuit with a circuit size n has $\sqrt{n}$ outputs.

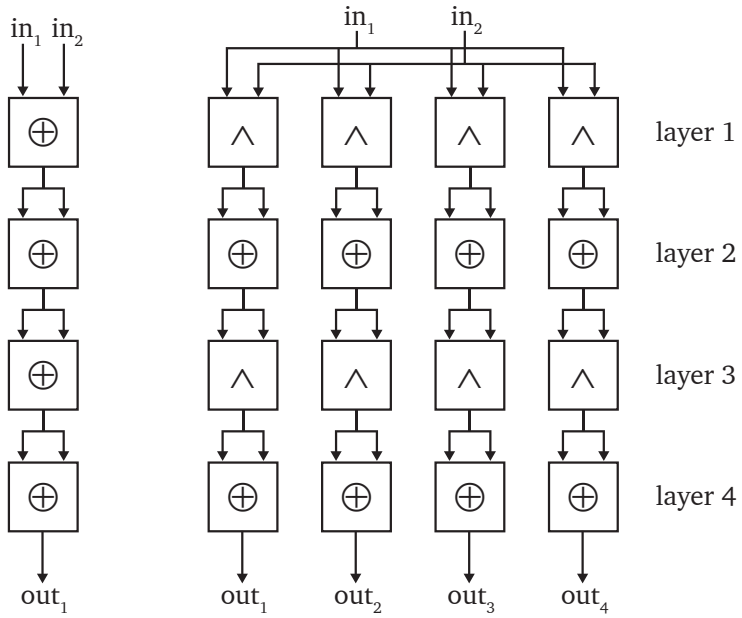


Figure 5.1: Circuit Structures. Example of a sequential circuit with four XOR gates (left) and a parallel circuit with alternating layers of AND gates and XOR gates and a total of $4 \times 4 = 16$ gates (right).

For the PFE setting, we use universal circuits which were generated using Kiss and Schneider's UC compiler [KS16] that implements Valiant's UC constructions. The simulated random circuits had circuit sizes 10, 100, 1 000 and 10 000. The sizes of the resulting universal circuits can be found in Table 5.1.

In 2PC protocols, X switches, Y switches and universal gates are usually implemented using AND and XOR gates. The evaluation of a Y switch requires the evaluation of two XOR gates and one AND gate while the evaluation of an X switch requires the evaluation of three XOR gates and one AND gate. Evaluating a universal gate requires the evaluation of three Y switches, i.e., the evaluation of nine gates in total, three of which are AND gates.
As implemented by the ABY framework, the GMW protocol uses this standard implementation while Yao's garbled circuit protocol uses a more efficient universal gate implementation with only one AND gate.

Table 5.1: Number of Gates in the UCs used for Benchmarking. Number of gates in the simulated circuit and resulting number of actual gates in the universal circuit for different gate types. The number of AND gates and XOR gates are computed assuming that X switches, Y switches and universal gates are implemented in the standard manner.

	#Gates in the Simulated Circuit			
	10	**100**	**1 000**	**10 000**
X Switches	42	1 678	32 132	482 656
Y Switches	11	167	1 191	11 807
Universal Gates	8	98	998	9 998
Total	61	1 943	34 321	504 461
AND Gates	77	2 139	36 317	524 457
XOR Gates	196	5 956	104 766	1 531 570

5.1.3 Protocol Phases

Our Intel SGX-based 2PC Protocol. We divide our 2PC protocol into two basic phases: the *remote attestation phase* and the actual *two-party computation phase* (see Protocol 4.1). The RA phase ends and the 2PC phase starts when the attestation ok message is received by the service provider. The 2PC phase includes the construction of the circuit message during which the inputs are read from file, encrypted and integrity-protected and the processing of the circuit message on the application side. It ends after the result message has been processed and the results have been written to file.

On the application side, we separately measure the time it takes to read the circuit file, to load it into the enclave, where it is parsed, and finally, to evaluate it inside the enclave. Less importantly, we measure the time it takes to load the service providers' private inputs into the enclave. This enables us to find out which steps in our current protocol implementation take the most time and need to be optimized.

Cryptographic 2PC Protocols. We compare our protocol to two cryptographic protocols: Yao's garbled circuit protocol and the GMW protocol as implemented by the ABY framework. Both protocols are divided into a *setup phase* and an *online phase*. The ABY framework is highly optimized for performance and includes the latest 2PC improvements. It's main design goal is to achieve an efficient online phase, which is why all cryptographic operations are pre-computed in the setup phase except for those required for evaluating garbled circuits. In Yao's protocol, the garbled circuit is transferred in the setup phase [DSZ15].
We furthermore compute the *total run-time* comprising the times for the setup phase and the online phase.

5.2 SGX Operations

This section contains benchmarking results for the enclave creation and for the remote attestation phase, which is independent of the two-party computation phase. The circuit has not yet been loaded into the enclave when remote attestation is performed. The run-time is therefore the same for both secure function evaluation and private function evaluation. The values we specify in this section are the overall average of 360 executions (2 circuit structures × 6 circuit sizes × 3 gate type compositions × 10 executions).

5.2.1 Enclave Creation

The enclave pages are cryptographically measured during enclave initialization, which is why the enclave creation time depends on the enclave's total size. We set the stack size to 8 KB and the heap size to 205 MB. Consequently, enclave creation via the function `sgx_create_enclave` takes an average time of 2 205 ms. The standard deviation was 13 ms.

We start the two service providers after the application, so that the enclave creation precedes the remote attestation phase.

5.2.2 Remote Attestation

The remote attestation phase comprises steps ① to ⑩ of our 2PC protocol (see Protocol 4.1). Both service providers individually perform remote attestation and establish a trusted channel with the same enclave.

Run-Time. Table 5.2 shows the average run-times we measured for the remote attestation phase in the LAN and WAN network settings.

Table 5.2: Remote Attestation Run-Time. Average run-time and corresponding standard deviation (SD) for the remote attestation phase in dependence on the network setting.

	Time [ms]	**SD** [ms]
LAN Setting	1 318.485	120.139
WAN Setting	2 612.672	104.920

Communication. In Table 5.3, we list the sizes of the plain messages that are exchanged between the service providers and the application during the remote attestation phase.

Table 5.3: Remote Attestation Plain Message Sizes. Plain message sizes in bytes for the messages sent from the service provider to the application (left) and the messages sent from the application to the service provider (right). The 20 byte header that each message is sent with is not included.

Service Provider → App		**App → Service Provider**	
request for attestation	2		
		msg0‖msg1	80
msg2	187		
		msg3	1471
attestation result	65		
platform information blob	103		
		attestation ok msg	4
total	357	**total**	1555

The actual communication for the remote attestation phase that was determined using Wireshark can be found in Table 5.4. It includes Ethernet, TCP/IP and TLS protocol overheads. Before the service provider and the application can communicate using TLS, a secure session has to be established. This is achieved with the help of the TLS Handshake Protocol. The total amount of data transmitted for the actual remote attestation process includes 12 TCP acknowledgement (ACK) packets, 6 message header packets and 6 message packets. The latter account for $642 + 1840 = 2482$ bytes out of the 3964 bytes that are exchanged.

Table 5.4: Remote Attestation Phase Communication. Amount of data exchanged between (a) the service provider and the application and (b) the service provider and the Intel Attestation Service (IAS).

	Communication [bytes]		
	SP → App	**App → SP**	**Total**
TLS Handshake	607	2 666	3 273
Remote Attestation	1 383	2 581	3 964
Total	1 990	5 247	7 237

(a) Communication between the Service Provider and the Application

	Communication [bytes]		
	SP → IAS	**IAS → SP**	**Total**
TLS Handshake	2 302	47 654	49 956
Remote Attestation	2 347	5 930	8 277
Total	4 649	53 584	58 233

(b) Communication between the Service Provider and the IAS

The amount of data exchanged between the service provider and the Intel Attestation Service is also contained in Table 5.4. We measured an average round trip time of 10.498 ± 0.785 ms for the connection to the IAS, which is not simulated and runs over Wi-Fi. The IAS uses *Mutual Transport Layer Security (MTLS)* for authentication. It transmits a very large amount of data during the TLS handshake. Generally, the amount of data exchanged between the service provider and the IAS is subject to fluctuations caused by variable-size packet segmentation and varying numbers of TCP ACK packets. The values in Table 5.4 are supposed to give an indication of the order of magnitude of the transmitted amount of data, both during the TLS handshake and during the actual remote attestation process. Specifically, the communication includes the retrieval of the (typically empty) signature revocation list and the verification of the quote which results in the attestation report being returned (see Protocol 2.1).

5.3 Intel SGX-Based Secure Two-Party Computation

In this section, we present our benchmarking results for the SFE setting and the PFE setting. Section 5.3.1 contains the results for the benchmarks run on the application side. We measure the time to read the circuit file, to load it into the enclave, to send the service providers' private inputs into the enclave and to evaluate the circuit inside the enclave. These benchmarks are independent of the network setting. Section 5.3.2 contains the benchmarking results for the two-party computation phase in the LAN and WAN network settings.

5.3.1 Application Benchmarks

Reading In the Circuit File. The application has to read in the circuit file in order to send it into the enclave. Since we re-ran our benchmarking experiments several times, the circuit file is cached and the values we specify in the following are for retrieving the circuit file from the cache. In the first benchmark run, the time required for reading in the circuit is typically much longer.

Table 5.5 contains the times required to read in different circuit files. They depend solely on the size of the circuit file and are independent of the gate type and network setting. In

the SFE setting, the values we specify are therefore an average of 60 executions. A parallel circuit has the same number of gates but more outputs than a sequential circuit of the same circuit size. The corresponding circuit file is therefore bigger and takes slightly longer to read in. In the PFE setting, the specified values are an average of 10 executions. Unexpectedly, we observed different times for the LAN and WAN network settings when starting the service provider providing the programming bits before the other service provider. In this case, the programming file is read before the circuit file. If the service providers are started in reverse order, the times observed for the LAN and WAN setting are the same. We assume this to be a caching phenomenon.

Table 5.5: Time Required for Reading In Different Circuit Files. Time needed to read in the circuit file representations of sequential and parallel Boolean circuits and universal circuits with different circuit sizes. The circuit sizes we specify for the PFE setting are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively.

	Sequential Circuits		**Parallel Circuits**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	0.098	0.015		
100	0.109	0.016	0.109	0.019
1 000	0.238	0.027		
10 000	2.100	0.232	2.126	0.206
100 000	13.965	5.866		
1 000 000	75.557	9.178	77.160	10.699

(a) SFE Setting

	LAN Setting		**WAN Setting**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	0.089	0.010	0.117	0.015
100	0.463	0.093	0.576	0.043
1 000	9.593	0.534	10.586	0.732
10 000	44.287	0.622	74.210	1.462

(b) PFE Setting

Loading the Circuit into the Enclave. The circuit file data is marshalled into the enclave, where it is parsed and hashed. Especially for circuits with large sizes this leads to a large amount of EPC paging and therefore to long run-times. In Table 5.6 the average times required to load circuits of different sizes into the enclave are summarized along with the corresponding standard deviations. A more detailed overview with the times to load circuits

with different circuit structures and gate types can be found in Appendix Section A.2.1. Because the times do not substantially differ, we specify overall average values at this point. In some cases, the times required for loading the circuit into the enclave do not seem to be uniformly distributed, which leads to large standard deviations. This was particularly observed for circuits with 10 000 gates, where in 105 of 120 executions an average time of 19.67 ± 0.64 ms was required and in the other 15 executions an average time of 45.21 ± 1.01 ms was required. We assume this to be due to different amounts of EPC paging. In cases like this one, Table 5.6 contains the average of the larger group of values, which is marked with an asterisk to point out that fact.
For the PFE setting, we specify the times measured for the LAN setting.

Table 5.6: Time Required for Loading Different Circuits into the Enclave. The time to parse and hash the circuit in-enclave is included. Values marked with an asterisk only cover the majority of executions but not all of them. The circuit sizes we specify for the PFE setting are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively.

Circuit Size	**Time** [ms]	**SD** [ms]
10	0.500	0.062
100	*0.913	*0.114
1 000	5.821	0.535
10 000	*19.665	*0.642
100 000	204.107	6.165
1 000 000	2 280.337	12.330

(a) SFE Setting

Circuit Size	**Time** [ms]	**SD** [ms]
10	0.732	0.068
100	*7.400	*0.158
1 000	92.843	1.606
10 000	1 205.211	1.727

(b) PFE Setting

Sending the Private Inputs into the Enclave. The service providers' private inputs are sent into the enclave along with the circuit hash. Inside the enclave, the CMAC of the circuit hash is verified and the private inputs are decrypted and unpacked.
We separately measure the time to send in the first and the second service provider's input and summarize them in Table 5.7. In the SFE setting, SP1 and SP2 both have a 1-bit private input. The same function is called to send that input into the enclave. Nonetheless, the required times differ measurably, presumably due to different amounts of EPC paging. SP1's input is sent into the enclave right after the circuit was loaded into the enclave. SP2's input is sent in afterwards. Due to the involved paging, the time also depends on the circuit size.
Again, for some circuit sizes, the times we observed were not uniformly distributed. For SP1's input and circuits with 1 000 000 gates, an average time of 250 ± 11 μs was observed in 98 of 120 executions and in the other 22 executions an average time of 851 ± 24 μs was observed. In cases like this one, we specify the more frequently occurring and typically smaller value in Table 5.7 and mark it with an asterisk.

In the PFE setting, SP2 has a 1-bit private input and SP1 has the programming to the universal circuit as private input, consisting of four bits per gate in the universal circuit. The universal circuits we use possess 61, 1 943, 34 321 and 504 461 gates, respectively (see Table 5.1). Again, we specify the times measured for the LAN setting at this point.

Looking at the values in Table 5.7, the large influence of EPC paging on enclave performance becomes apparent. While in the SFE setting, the time required to send SP2's private input into the enclave is almost constant, it observably differs from the time required to send SP1's private input into the enclave. For SP1 the required time depends on the size of the circuit that was loaded into the enclave right before the input is sent in.
In the PFE setting, the length of SP1's private input increases with the circuit size. For large circuit sizes, the time required to send this input into the enclave also notably increases. For smaller circuit sizes, this effect is masked by EPC paging effects.

Table 5.7: Time Required for Sending the Private Inputs into the Enclave. Time needed to separately send both service providers' private inputs into the enclave in dependence on the size of the circuit that was previously loaded into the enclave. Values marked with an asterisk only cover the majority of executions but not all of them. The circuit sizes we specify for the PFE setting are the sizes of the simulated circuits.

	SP1's Input		**SP2's Input**	
Circuit Size	**Time** [μs]	**SD** [μs]	**Time** [μs]	**SD** [μs]
10	112	15	86	5
100	*116	*19	88	8
1 000	*39	*5	85	4
10 000	54	4	89	11
100 000	55	2	87	11
1 000 000	*250	*11	73	13

(a) SFE Setting

	SP1's Input		**SP2's Input**	
Circuit Size	**Time** [μs]	**SD** [μs]	**Time** [μs]	**SD** [μs]
10	115	13	94	3
100	*52	*11	95	2
1 000	185	3	94	3
10 000	2 314	35	89	6

(b) PFE Setting

Evaluating the Circuit inside the Enclave. When both service provider's private inputs are available, the circuit is evaluated inside the enclave as described in Section 4.2.1. Beforehand, the three circuit hash values are checked for equality. Afterwards, the computation result is encrypted for the service provider requesting it and marshalled out of the enclave.
Table 5.8 contains the average times required for evaluating sequential and parallel Boolean circuits and universal circuits of different sizes. Since parallel circuits have more outputs that need to be encrypted and marshalled out of the enclave, their evaluation takes slightly longer compared to sequential circuits. Because the times do not substantially differ for circuits with different gate types, we specify overall average values at this point. For the SFE setting, a more detailed overview can be found in Appendix Section A.2.1.

Table 5.8: Time Required for Evaluating Different Circuits inside the Enclave. The time to check the three circuit hash values for equality and to encrypt the computation result and marshal it out of the enclave is included. Values marked with an asterisk only cover the majority of executions but not all of them. The circuit sizes we specify for the PFE setting are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively.

	Sequential Circuits		**Parallel Circuits**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	0.173	0.012		
100	0.340	0.023	0.339	0.033
1 000	2.222	0.040		
10 000	*9.462	*0.522	*9.613	*0.444
100 000	*59.163	*2.650		
1 000 000	1 901.219	6.598	1 933.360	10.291

(a) SFE Setting

Circuit Size	**Time** [ms]	**SD** [ms]
10	0.320	0.019
100	6.207	0.651
1 000	34.800	0.290
10 000	1 145.774	2.032

(b) PFE Setting

5.3.2 Two-Party Computation Phase

As the application in our Intel SGX-based 2PC protocol was originally designed to support only one service provider, it is single-threaded. Both the remote attestation and the two-party computation phase will therefore take substantially longer if both service providers are started at the same time. For this reason, we start the second service provider (SP2) 6 seconds after the first (SP1). Nonetheless, we also present experimental results for starting SP1 and SP2 at the same time for comparison. They can be found in Appendix Section A.2.1.
Our 2PC protocol does not require direct interaction between the two service providers. After performing remote attestation, they merely both need to send their inputs to the enclave, from which they can later obtain the result. Currently, the circuit is loaded into the enclave when the first circuit message is received. SP1's two-party computation phase therefore includes this step. As the second service provider's input is not yet available, SP1 receives a result not ready message and is sent to sleep for 8 seconds. During that time, SP2 performs remote attestation and transmits its inputs. The circuit is evaluated and the computation result is returned to SP2. When SP1 awakes, it can obtain the result from the enclave.
In the PFE setting, we only return the computation result to SP2 and stop SP1 after it has received the result not ready message. We want to note that the enclave will not return the result to SP1 even if it should request it. This protects SP2's private input from SP1 that can make the universal circuit compute an arbitrary function up to a given size.

SP1's sleep duration was set to be this long to fit both the LAN and WAN network settings and to be able to exclude any interference between the two service providers. Our implementation could of course be improved upon by adding support for multi-threading on the application side and/or by loading the circuit into the enclave during a distinct setup phase.

Run-Time in the SFE Setting. Table 5.9 contains the run-time results for the two-party computation phase in the LAN and WAN network settings. As explained in the previous section, some of the run-times on the application side are not uniformly distributed. In some cases, this also leads to non-uniform run-time results on the service provider side. In those case, we specify the more frequently occurring value and mark it with an asterisk.

Table 5.9: Two-Party Computation Run-Time in the SFE Setting. Average run-time for the two-party computation phase in (a) the LAN setting and (b) the WAN setting. The time required by SP1 to send its input includes the time to load the circuit into the enclave and ends when the result not ready message is received. The time required to get the result is started when SP1 awakes after sleeping for 8 seconds and includes the processing of the result. The time required by SP2 includes the evaluation of the circuit.

	SP1				**SP2**	
	Send Input		**Get Result**		**Complete 2PC Phase**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	88.605	1.294	46.667	0.138	89.049	1.361
100	89.934	2.116	46.682	0.426	89.010	1.119
1 000	95.156	1.963	46.624	0.082	93.014	1.383
10 000	115.777	12.022	*46.704	*0.107	99.561	5.222
100 000	314.957	12.244	5.229	0.133	162.523	13.814
1 000 000	2 447.729	18.916	4.845	0.223	2 006.275	18.321

(a) LAN Setting

	SP1				**SP2**	
	Send Input		**Get Result**		**Complete 2PC Phase**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	380.709	14.680	346.098	6.797	386.889	7.971
100	380.345	16.334	350.050	27.225	386.507	9.228
1 000	390.303	8.122	*343.827	*6.464	383.181	15.907
10 000	408.685	9.487	*345.131	*11.694	397.022	11.337
100 000	598.326	9.814	301.518	7.365	448.012	9.099
1 000 000	2 742.420	15.172	306.603	23.578	2 303.766	19.317

(b) WAN Setting

Run-Time in the PFE Setting. In Table 5.10 the run-time results for the two-party computation phase in the LAN and WAN network settings are summarized.

Table 5.10: Two-Party Computation Run-Time in the PFE Setting. Average run-time for the two-party computation phase in (a) the LAN setting and (b) the WAN setting. The time required by SP1 to send its input includes the time to load the circuit into the enclave and ends when the result not ready message is received. The time required by SP2 includes the evaluation of the circuit. Values marked with an asterisk only cover the majority of executions but not all of them. The circuit sizes we specify are the sizes of the simulated circuits.

	SP1		**SP2**	
	Send Input		**Complete 2PC Phase**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	88.460	0.996	88.687	1.232
100	95.141	1.243	96.507	2.437
1 000	199.026	2.969	124.684	1.201
10 000	1328.588	8.593	1237.075	9.607

(a) LAN Setting

	SP1		**SP2**	
	Send Input		**Complete 2PC Phase**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	361.104	21.163	401.543	39.310
100	391.743	9.955	392.385	4.318
1 000	481.876	19.984	419.033	10.600
10 000	1855.970	16.570	1530.428	6.952

(b) WAN Setting

Communication in the SFE Setting. In the SFE setting, the amount of data sent by the service providers to the application does not depend on the circuit size or the circuit structure. SP2 only sends the circuit message which has a size of 81 bytes for a 1-bit input. SP1 additionally sends a result request message, transmitting $81 + 4 = 85$ bytes in total.
The size of the result message sent by the application depends on the circuit structure. All sequential circuits have a single output while a parallel circuit with circuit size n has $\sqrt{n}$ output bits. When a sequential circuit is evaluated, the application transmits 27 bytes to SP2. It additionally transmits a result not ready message to SP1, leading to a total of $27+4 = 31$ bytes. The amount of plain message data transmitted by the application to the two service providers in case a parallel circuit is evaluated is summarized in Table 5.11. Additionally, a 20 byte header is transmitted per message.

Table 5.11: Plain Message Data Exchanged in the 2PC Phase for Parallel Circuits. Amount of plain message data transmitted by the application to the two service providers SP1 and SP2 when parallel circuits of different sizes are computed in the SFE setting. Header data is not included.

Circuit Size	Plain Message Data [bytes]			
	SP1 → App	SP2 → App	App → SP1	App → SP2
100	81	85	32	28
10 000	81	85	43	39
1 000 000	81	85	156	152

The actual communication for the two-party computation phase that was determined using Wireshark can be found in Table 5.12. It includes Ethernet, TCP/IP and TLS protocol overheads. For sequential circuits, the application transmits 369 bytes to SP1 and 715 bytes to SP2. The amount of data transmitted by SP1 and SP2 to the application is the same as for parallel circuits.

Table 5.12: Communication in the 2PC Phase for Parallel Circuits. Amount of data exchanged between the application and the two service providers SP1 and SP2 when parallel circuits of different sizes are computed in the SFE setting.

Circuit Size	Communication [bytes]			
	SP1 → App	SP2 → App	App → SP1	App → SP2
100	423	769	370	716
10 000	423	769	381	727
1 000 000	423	769	494	840

Communication in the PFE Setting. In the PFE setting, the amount of data sent by SP2 is constant while the amount of data sent by SP1 to the application depends on the circuit size as the number of programming bits increases with the number of gates. This can be seen in Table 5.13. For large circuits, the private input is split over multiple TCP packets. This also leads to an increase in the total amount of data sent by the application to SP1, which includes ACK packets.
The universal circuits we use all have a single output bit, so that the size of the result message sent by the application to SP2 has a constant size. SP1 does not receive the computation result.

Table 5.13: Communication in the 2PC Phase for Universal Circuits. Amount of data exchanged between the application and the two service providers SP1 and SP2 in the PFE setting. The circuit sizes we specify are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively.

Circuit Size	**Communication** [bytes]			
	SP1 → App	**SP2 → App**	**App → SP1**	**App → SP2**
10	453	423	346	369
100	1 396	423	346	369
1 000	17 616	423	346	369
10 000	230 198	423	1 078	369

5.4 ABY-Based Secure Two-Party Computation

In this section, we present our benchmarking results for Yao's garbled circuit protocol and the GMW protocol as implemented by the ABY framework in the SFE setting and the PFE setting. In Section 5.4.1, we summarize the run-time for the base-OTs. The following Sections 5.4.2ff. contain the times for the setup phase and the online phase and the total time. We present experimental results for the LAN and WAN network settings. The values we specify are for 128 bit symmetric security.

5.4.1 Base-OTs

OT extension protocols use a small number of base-OTs to cheaply compute a very large number of OTs. The base-OTs are used in both Yao's garbled circuit protocol and the GMW protocol and need to be performed once when the connection between the client and the server is established. Because the time for the base-OTs is a one-time expense, it is not included in the setup time nor the total time.

Table 5.14 shows the average run-times we measured for the base-OTs in the LAN and WAN network settings. The specified values are an average of 360 executions. 49 956 bytes are sent (and also received) by both the client and the server.

Table 5.14: Base-OT Run-Time. Average run-time and corresponding standard deviation for the base-OTs in dependence on the network setting.

	Time [ms]	**SD** [ms]
LAN Setting	377.595	41.557
WAN Setting	561.591	56.623

5.4.2 Setup Phase

Yao's Garbled Circuit Protocol. In the constant-round setup phase of Yao's protocol, all cryptographic operations are pre-computed except for those required for evaluating the garbled circuit. Furthermore, the garbled circuit is transferred. The run-time results for the setup phase in the LAN and WAN network settings are summarized in Table 5.15. The server acts as the circuit garbler and the client acts as the circuit evaluator. The specified values are for the client. A complete overview of the average setup phase run-times for the client and the server along with the corresponding standard deviations can be found in Appendix Section A.2.2.

Table 5.15: Setup Phase Run-Time for Yao's Protocol in the SFE Setting. Average run-time for the setup phase when computing sequential and parallel circuits in (a) the LAN setting and (b) the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	1.637		1.661		1.696	
100	1.711	1.738	1.756	1.752	1.762	1.759
1 000	1.707		1.976		2.307	
10 000	2.339	2.298	4.973	4.978	7.717	7.652
100 000	6.784		32.801		60.384	
1 000 000	51.350	52.632	205.208	207.551	342.664	349.783

(a) LAN Setting

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	107.255		109.304		110.240	
100	108.281	108.639	108.637	108.916	109.727	109.665
1 000	108.450		110.119		110.178	
10 000	111.512	108.553	203.177	203.102	205.900	204.123
100 000	128.175		419.272		648.742	
1 000 000	199.807	221.747	1 871.232	1 885.767	3 482.470	3 472.997

(b) WAN Setting

For Yao's protocol, the run-time and communication does not depend on the AND depth of the circuit. The run-times and communication for sequential and parallel circuits are therefore similar. Since parallel circuits have more outputs, they take slightly longer to compute. This is only reflected in the results for large circuits.

Table 5.16 contains the setup phase communication for the client. For 128 bit symmetric security $2 \times 128 = 256$ bits $= 32$ bytes have to be transmitted per AND gate in the circuit.

Table 5.16: Setup Phase Communication for Yao's Protocol in the SFE Setting. Amount of data that is sent and received by the client during the setup phase of Yao's protocol when sequential and parallel circuits of different sizes are computed. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sent	**Rcv**	**Sent**	**Rcv**	**Sent**	**Rcv**
10	2 082	44	2 082	204	2 082	364
100	2 082	44	2 082	1 644	2 082	3 244
1 000	2 082	44	2 082	16 044	2 082	32 044
10 000	2 082	44	2 082	160 044	2 082	320 044
100 000	2 082	44	2 082	1 600 044	2 082	3 200 044
1 000 000	2 082	44	2 082	16 000 071	2 082	32 000 107

(a) Sequential Circuits

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sent	**Rcv**	**Sent**	**Rcv**	**Sent**	**Rcv**
100	2 082	45	2 082	1 645	2 082	3 245
10 000	2 082	56	2 082	160 056	2 082	320 056
1 000 000	2 082	168	2 082	16 000 195	2 082	32 000 231

(b) Parallel Circuits

The GMW Protocol. In the GMW protocol, all required symmetric cryptographic computations are performed during the setup phase, which has a constant number of rounds. This includes the pre-computation of the multiplication triples for the AND gates. The ABY framework implements load balancing for the setup phase of the GMW protocol. As a consequence,

the work and communication is equally distributed among the two parties. We therefore report an average of the run-times for the client and the server in Table 5.17.

For the GMW protocol, the run-time and communication depend on the AND depth of the circuit. The run-time and communication for a parallel circuit is therefore substantially smaller than for the sequential circuits of the same AND size.

We set the limit for the end-to-end run-time to 10 minutes = 600 000 milliseconds. The table contains an upward pointing arrow (↑) if the computation was not completed during that time and thus, no timing result was obtained.

Table 5.17: Setup Phase Run-Time for the GMW Protocol in the SFE Setting. Average run-time for the setup phase when computing sequential and parallel circuits in (a) the LAN setting and (b) the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	0.104		1.558		1.605	
100	0.098	0.091	1.698	1.574	1.949	1.576
1 000	0.097		4.852		7.895	
10 000	0.082	0.096	34.343	5.360	63.423	9.358
100 000	0.081		175.645		↑	
1 000 000	0.088	0.071	1 606.825	208.181	↑	378.451

(a) LAN Setting

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	0.172		105.449		106.395	
100	0.170	0.185	106.123	105.472	107.707	106.394
1 000	0.171		211.329		217.556	
10 000	0.179	0.181	445.263	211.187	↑	222.847
100 000	0.173		↑		↑	
1 000 000	0.161	0.181	↑	1 796.707	↑	3 185.629

(b) WAN Setting

The communication for the setup phase of the GMW protocol can be found in Table 5.18. It is equal for client and server. Per AND gate, one multiplication triple is required and per multiplication triple, 134 bits have to be transferred. The multiplication triples for all AND gates of one layer are transferred together. The resulting number of bits is padded to a multiple of 8.

Table 5.18: Setup Phase Communication for the GMW Protocol in the SFE Setting. Amount of data that is sent (and also received) by both the client and the server during the setup phase of the GMW protocol when sequential and parallel circuits of different sizes are computed. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	0		2 091		2 091	
100	0	0	8 235	2 091	16 419	4 139
1 000	0		73 771		145 451	
10 000	0	0	720 939	88 107	1 441 835	174 123
100 000	0		7 200 904		↑	
1 000 000	0	0	72 002 429	8 065 142	↑	16 130 266

PFE Setting. The benchmarking results for the setup phase of both Yao's garbled circuit protocol and the GMW protocol in the PFE setting are presented in the following two tables. Table 5.19 contains the average run-times in the LAN and WAN network settings and Table 5.20 contains the communication.

Table 5.19: Setup Phase Run-Time in the PFE Setting. Average run-time for the setup phase of Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed.

Circuit Size	**Time** [ms]			
	Yao		**GMW**	
	LAN	**WAN**	**LAN**	**WAN**
10	2.476	199.262	1.715	106.235
100	3.765	198.711	5.410	212.564
1 000	32.120	356.794	53.141	549.132
10 000	393.929	2 057.248	353.315	↑

Table 5.20: Setup Phase Communication in the PFE Setting. Amount of data sent and received by the client during the setup phase of Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed.

Circuit Size	Communication [bytes]		
	Yao		GMW
	Sent	Rcv	Sent/Rcv
10	2 082	3 102	6 187
100	2 082	63 782	94 251
1 000	2 082	1 114 286	1 187 883
10 000	2 082	16 302 784	14 471 361

5.4.3 Online Phase

Yao's Garbled Circuit Protocol. Yao's protocol has a constant-round online phase. The evaluation of the garbled circuit is performed non-interactively by the client. Symmetric cryptography is required per AND gate. The online phase run-time therefore increases with the circuit's AND size. The run-times for sequential and parallel Boolean circuits with different gate types and circuit sizes are summarized in Table 5.21. Again, we report the run-times for the client.

Table 5.21: Online Phase Run-Time for Yao's Protocol in the SFE Setting. Average run-time for the online phase when computing sequential and parallel circuits in (a) the LAN setting and (b) the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	Time [ms]					
	X		AX		A	
	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
10	2.652		2.583		2.548	
100	2.617	2.624	2.687	2.580	2.584	2.619
1 000	2.677		2.656		2.876	
10 000	3.428	3.445	3.963	4.539	4.613	5.044
100 000	10.327		14.670		19.360	
1 000 000	79.532	80.360	117.649	118.346	163.657	171.337

(a) LAN Setting

Circuit Size	Time [ms]					
	X		AX		A	
	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
10	213.836		210.284		212.144	
100	212.250	211.892	210.180	211.147	208.809	211.081
1 000	214.182		214.356		211.817	
10 000	213.615	215.332	217.766	215.740	218.848	219.915
100 000	242.127		232.797		229.768	
1 000 000	304.553	305.056	338.923	340.471	390.611	381.711

(b) WAN Setting

The communication for the online phase can be found in Table 5.22. During the online phase, the server transmits its garbled private inputs to the client. The client receives its garbled private inputs via oblivious transfer. It then evaluates the circuit gate by gate. Finally, it sends the computation result to the server. The amount of data sent by the client in the online phase is therefore slightly larger for parallel circuits than for sequential circuits.

Table 5.22: Online Phase Communication for Yao's Protocol in the SFE Setting. Amount of data that is sent and received by the client during the online phase of Yao's protocol when sequential and parallel circuits of different sizes are computed.

Circuit Size	Communication [bytes]			
	Sequential		Parallel	
	Sent	Rcv	Sent	Rcv
10	29	50		
100	29	50	30	50
1 000	29	50		
10 000	29	50	41	50
100 000	29	50		
1 000 000	29	50	153	50

The GMW Protocol. Table 5.23 contains the average run-time results for the online phase of the GMW protocol, during which two independent 2-bit messages have to be transmitted per layer of AND gates. This need for interaction constitutes a performance bottleneck, especially in high latency networks. The evaluation of the circuit only requires One Time Pad operations.

Table 5.23: Online Phase Run-Time for the GMW Protocol in the SFE Setting. Average run-time for the online phase when computing sequential and parallel circuits in (a) the LAN setting and (b) the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	2.068		4.988		7.933	
100	2.058	2.046	31.583	4.983	61.175	7.962
1 000	2.092		368.193		778.928	
10 000	2.904	2.943	3 795.195	33.861	7 193.462	63.310
100 000	11.962		34 909.385		↑	
1 000 000	97.303	104.492	365 021.800	504.003	↑	1 008.987

(a) LAN Setting

Circuit Size	**Time** [ms]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	159.505		424.059		689.449	
100	158.402	158.578	2 822.847	425.496	5 475.791	686.835
1 000	160.217		26 761.325		53 325.215	
10 000	160.433	162.258	266 182.750	2 827.786	↑	5 484.357
100 000	189.730		↑		↑	
1 000 000	295.079	310.269	↑	27 272.910	↑	54 423.600

(b) WAN Setting

The online phase communication is summarized in Table 5.24. The communication complexity of the GMW protocol depends on the AND size and AND depth of the circuit.

Table 5.24: Online Phase Communication for the GMW Protocol in the SFE Setting. Amount of data that is sent (and also received) by both the client and the server during the online phase of the GMW protocol when sequential and parallel circuits of different sizes are computed. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	29		84		139	
100	29	30	579	95	1 129	160
1 000	29		5 529		11 029	
10 000	29	41	55 029	1 791	110 029	3 541
100 000	29		550 029		↑	
1 000 000	29	153	5 500 029	129 653	↑	259 153

PFE Setting. The benchmarking results for the setup phase of both Yao's garbled circuit protocol and the GMW protocol in the PFE setting are presented in the following two tables. Table 5.19 contains the average run-times in the LAN and WAN network settings and Table 5.20 contains the communication.

Table 5.25: Online Phase Run-Time in the PFE Setting. Average run-time for the online phase of Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed.

Circuit Size	**Time** [ms]			
	Yao		**GMW**	
	LAN	**WAN**	**LAN**	**WAN**
10	2.522	212.556	22.928	2 016.336
100	3.588	214.722	328.720	24 738.735
1 000	21.668	256.743	3 568.120	252 157.050
10 000	364.890	1 066.834	31 914.340	↑

Table 5.26: Online Phase Communication in the PFE Setting. Amount of data sent and received by the client during the online phase of Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed. The programming bits to the universal circuit are provided by the server.

Circuit Size	Communication [bytes]			
	Yao		GMW	
	Sent	Rcv	Sent	Rcv
10	29	41	414	421
100	29	31 146	5 345	5 587
1 000	29	549 177	58 102	62 392
10 000	29	8 071 426	622 262	685 319

5.4.4 Total Time & Communication

The total time comprises the times for the setup phase and the online phase. The total run-time results for Yao's protocol are shown alongside the results for the GMW protocol in Table 5.27. They are for computing sequential circuits and parallel circuits in the SFE setting.

Table 5.27: Total Run-Time for Sequential Circuits. Average total run-time for Yao's protocol and the GMW protocol when computing (a) sequential circuits in the LAN setting, (b) sequential circuits in the WAN setting, (c) parallel circuits in the LAN setting and (d) parallel circuits in the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	Time [ms]					
	X		AX		A	
	Yao	GMW	Yao	GMW	Yao	GMW
10	4.288	2.172	4.244	6.546	4.244	9.538
100	4.328	2.156	4.443	33.282	4.345	63.125
1 000	4.384	2.189	4.632	373.045	5.183	786.823
10 000	5.767	2.986	8.936	3 829.537	12.330	7 256.885
100 000	17.111	12.043	47.471	35 085.030	79.744	↑
1 000 000	130.882	97.391	322.857	366 628.625	506.321	↑

(a) Sequential Circuits + LAN Setting

Circuit Size	Time [ms]					
	X		AX		A	
	Yao	GMW	Yao	GMW	Yao	GMW
10	321.091	159.677	319.588	529.508	322.384	795.844
100	320.531	158.573	318.816	2 928.970	318.535	5 583.497
1 000	322.632	160.388	324.474	26 972.654	321.995	53 542.771
10 000	325.126	160.612	420.943	266 628.013	424.747	↑
100 000	370.302	189.903	652.070	↑	878.510	↑
1 000 000	504.360	295.240	2 210.155	↑	3 873.081	↑

(b) Sequential Circuits + WAN Setting

Circuit Size	Time [ms]					
	X		AX		A	
	Yao	GMW	Yao	GMW	Yao	GMW
100	4.362	2.137	4.332	6.557	4.377	9.538
10 000	5.743	3.040	9.517	39.221	12.696	72.668
1 000 000	132.992	104.563	325.897	712.184	521.120	1 387.439

(c) Parallel Circuits + LAN Setting

Circuit Size	Time [ms]					
	X		AX		A	
	Yao	GMW	Yao	GMW	Yao	GMW
100	320.531	158.762	320.062	530.968	320.746	793.228
10 000	323.885	162.439	418.843	3 038.973	424.039	5 707.204
1 000 000	526.802	310.450	2 226.238	29 069.617	3 856.143	57 609.229

(d) Parallel Circuits + WAN Setting

The GMW protocol requires interaction for the secure evaluation of AND gates. It performs well for circuits with low AND sizes and AND depths. The run-times for parallel circuits are therefore substantially shorter than for sequential circuits. In contrast, Yao's protocol only depends on the circuit's AND size, resulting in similar run-times for sequential and parallel circuits.

Table 5.28 contains the total communication for Yao's protocol and the GMW protocol in the SFE setting.

Table 5.28: Total Per-Party Communication in the SFE Setting. Total amount of data that is sent and received by the client (a) in Yao's protocol when sequential circuits are being evaluated, (b) in Yao's protocol when parallel circuits are being evaluated and (c) in the GMW protocol. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sent	**Rcv**	**Sent**	**Rcv**	**Sent**	**Rcv**
10	2 011	94	2 011	254	2 011	414
100	2 011	94	2 011	1 694	2 011	3 294
1 000	2 011	94	2 011	16 094	2 011	32 094
10 000	2 011	94	2 011	160 094	2 011	320 094
100 000	2 011	94	2 011	1 600 094	2 011	3 200 094
1 000 000	2 011	94	2 011	16 000 121	2 011	32 000 157

(a) Yao's Protocol + Sequential Circuits

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sent	**Rcv**	**Sent**	**Rcv**	**Sent**	**Rcv**
100	2 112	95	2 112	1 695	2 112	3 295
10 000	2 123	106	2 123	160 106	2 123	320 106
1 000 000	2 235	218	2 235	16 000 245	2 235	32 000 281

(b) Yao's Protocol + Parallel Circuits

Circuit Size	**Communication** [bytes]					
	X		**AX**		**A**	
	Sequential	**Parallel**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
10	29		2 175		2 230	
100	29	30	8 814	2 186	17 556	4 299
1 000	29		79 300		156 480	
10 000	29	41	775 968	89 898	1 551 864	177 664
100 000	29		7 750 933		↑	
1 000 000	29	153	77 502 458	8 194 795	↑	16 389 419

(c) GMW Protocol

PFE Setting. The circuit sizes we specify for the PFE setting are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively. Table 5.29 contains the total run-times for Yao's protocol and the GMW protocol.

Table 5.29: Total Run-Time in the PFE Setting. Average total run-time for Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed.

Circuit Size	Time [ms]			
	Yao		GMW	
	LAN	WAN	LAN	WAN
10	4.998	411.818	24.643	2 122.571
100	7.353	413.432	334.130	24 951.299
1 000	53.788	613.537	3 621.261	252 706.182
10 000	758.819	3 124.082	32 270.409	↑

Table 5.30: Total Per-Party Communication in the PFE Setting. Total amount of data sent and received by the client during Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed. The programming bits to the universal circuit are provided by the server.

Circuit Size	Communication [bytes]			
	Yao		GMW	
	Sent	Rcv	Sent	Rcv
10	2 111	3 143	6 601	6 608
100	2 111	94 928	99 596	99 838
1 000	2 111	1 663 463	1 245 985	1 250 275
10 000	2 111	24 374 210	15 093 632	15 156 680

5.5 Discussion & Comparison

In this section, we discuss the benchmarking results presented in the preceding sections and we directly compare our Intel SGX-based solution for secure function evaluation and private function evaluation to an ABY-based cryptographic solution.

5.5.1 One-Time Expenses

The base-OTs used in Yao's garbled circuit protocol and the GMW protocol need to be performed once when the connection between the client and the server is established. They are a one-time expense and therefore not counted towards the total time.
We argue that the remote attestation phase of our Intel SGX-based 2PC protocol is actually similar to the base-OTs. Both service providers need to individually perform remote attestation in order to establish a trusted channel with the same enclave. In theory, once this has been done, they can use the same enclave for computing many different circuits. The enclave could even use sealing to persistently store the secret keys that were established with the service providers. The remote attestation phase would not have to be performed again unless the enclave identity (enclave measurement) changed.
Table 5.31 contains the run-time and communication for both the remote attestation phase and the base-OTs for comparison. The RA phase takes considerably longer, although due to our set-up, they are hard to compare. The remote attestation run-time is mainly dictated by the speed of the connection with the Intel Attestation Service, which we did not simulate, and by the amount of data transferred over it, which we cannot optimize.
For the RA phase, we report the amount of data sent and received by SP1 and SP2 each. This includes the communication with the IAS. We additionally report the total amount of data sent and received by the application, which is for both service providers. For the base-OTs, we report the amount of data sent and received by the client and server each.

Table 5.31: One-Time Expenses Run-Times. (a) Average run-times and (b) communication for the remote attestation phase and for the base-OTs.

	Time [ms]	
	RA	**Base-OTs**
LAN Setting	1 318.485	377.595
WAN Setting	2 612.672	561.591

(a) Run-Time

	Communication [bytes]		
	RA		**Base-OTs**
	SP1/SP2	**App**	**Client/Server**
Sent	6 639	10 494	49 956
Rcv	58 831	3 980	49 956

(b) Communication

5.5.2 Two-Party Computation Phase

Our Intel SGX-based 2PC protocol does not require direct interaction between the two service providers. They both send their encrypted private inputs into the same enclave, within which the circuit is computed and from which they can later request the result. As the application and enclave are single-threaded, we execute the two service-providers, SP1 and SP2, effectively consecutively to rule out any interference between the two. SP1 is started 6 seconds before SP2 and is sent to sleep for 8 seconds after it has received a result not ready message in response to transmitting its inputs.

Currently, the circuit is read and loaded into the enclave after SP1's circuit message has been received. The time required for this process is therefore included in SP1's two-party computation phase. We argue that our implementation could be improved by adding support for multi-threading on the application side and/or by loading the circuit into the enclave during a distinct setup phase, which is run before SP1 and SP2 are started. To support this theory, we compute the time theoretically required by SP1 for the two-party computation phase if the circuit did not have to be read and loaded into the enclave. The results can be found in Table 5.32. We additionally compute the time required by SP2 for the two-party computation phase excluding the circuit evaluation. We report average values for the evaluation of sequential and parallel circuits and do not exclude EPC paging-related outliers. It should however be noted that for circuits with 1 million gates, the run-times for the evaluation of sequential and parallel circuits differ by ∼30 ms (see Appendix Section A.2.1), which is also reflected in the run-time for SP2's 2PC phase. This is because parallel circuits have more outputs, which need to be computed and encrypted. To simplify the comparison, we disregard this <1%-deviation at this point. For larger circuit sizes, the two cases should be differentiated.

PFE Setting. Analogously to the SFE setting, we compute the time theoretically required by SP1 for the two-party computation phase if the circuit did not have to be read and loaded into the enclave. The results can be found in Table 5.33. Since we do not return the computation result to SP1 in the PFE setting, SP1's 2PC phase ends after the result not ready message is received. As we reuse the implementation for the SFE setting, the application still performs an ECALL to try and obtain the result after SP1's circuit hash and private inputs have been sent into the enclave. This unnecessary step could be removed for a small run-time improvement.

Table 5.32: Two-Party Computation Run-Time in the SFE Setting. Average run-time for the two-party computation phase (a) for SP1 in the LAN setting, (b) for SP1 in the WAN setting and (c) for SP2 in the LAN and WAN settings. The time required by SP1 to send its input includes the time to read the circuit and to load it into the enclave and ends when the result not ready message is received. The column labelled "Δ" contains the difference of those times. The time required to get the result is started when SP1 awakes after sleeping for 8 seconds and includes the processing of the result. The time required by SP2 includes the time for the evaluation of the circuit. In the columns "ΔLAN" and "ΔWAN" this time has been subtracted.

Circuit Size	Send Input	Read+Load	Δ	Get Result
10	88.605	0.598	88.007	46.667
100	89.934	1.379	88.555	46.682
1 000	95.156	6.060	89.096	46.624
10 000	115.777	25.001	90.776	46.704
100 000	314.957	218.072	96.885	5.229
1 000 000	2 447.729	2 355.246	92.483	4.845

(a) SP1 + LAN Setting

Circuit Size	Send Input	Read+Load	Δ	Get Result
10	380.709	0.598	380.111	346.098
100	380.345	1.379	378.966	350.050
1 000	390.303	6.060	384.243	343.827
10 000	408.685	25.001	383.684	345.131
100 000	598.326	218.072	380.254	301.518
1 000 000	2 742.420	2 355.246	387.174	306.603

(b) SP1 + WAN Setting

	2PC Phase				
Circuit Size	**LAN**	**WAN**	**Evaluate**	**Δ LAN**	**Δ WAN**
10	89.049	386.889	0.173	88.876	386.716
100	89.010	386.507	0.340	88.670	386.167
1 000	93.014	383.181	2.222	90.792	380.959
10 000	99.561	397.022	10.591	88.970	386.431
100 000	162.523	448.012	63.883	98.640	384.129
1 000 000	2 006.275	2 303.766	1 917.289	88.986	386.477

(c) SP2

Table 5.33: Two-Party Computation Run-Time in the PFE Setting. Average run-time for the two-party computation phase in (a) the LAN setting and (b) the WAN setting. The time required by SP1 to send its input includes the time to read the circuit and to load it into the enclave and ends when the result not ready message is received. SP1 provides the programming to the universal circuit. The column labelled "Δ" contains the difference of those times. The time required by SP2 includes the time for the evaluation of the circuit.

	SP1			SP2	Total
Circuit Size	**Send Input**	**Read+Load**	Δ	**2PC Phase**	Σ
10	88.460	0.830	87.630	88.687	176.317
100	95.141	7.863	87.278	96.507	183.785
1 000	199.026	102.436	96.590	124.684	221.274
10 000	1 328.588	1 249.498	79.090	1 237.075	1 316.165

(a) LAN Setting

	SP1			SP2	Total
Circuit Size	**Send Input**	**Read+Load**	Δ	**2PC Phase**	Σ
10	364.104	0.962	363.142	388.537	751.680
100	391.743	8.111	383.632	392.385	776.016
1 000	481.876	105.658	376.218	419.033	795.251
10 000	1 855.970	1 281.562	574.408	1 530.428	2 104.836

(b) WAN Setting

5.5.3 SFE Setting

Before we directly compare the run-times and communication of our Intel SGX-based 2PC protocol with that of Yao's garbled circuit protocol and the GMW protocol, we describe our considerations for the comparison. This section deals with the SFE setting while the next one deals with the PFE setting.

The total time for Yao's protocol and the GMW protocol does not include the time to read and construct the circuit from file or the time for the base-OTs. We therefore think, it is reasonable to compare it to the time required by SP2 for the two-party computation phase, which does not contain the time to read the circuit or to load it into the enclave, nor the remote attestation time.
For SP1, the comparison is more difficult. In the current implementation of our protocol, the total time required by SP1 is extended due to its long sleep duration, which could however theoretically be used to perform other work. Additionally, the time to read and load the circuit into the enclave is included. For circuits with 1 000 000 gates, this time is quite large with approximately 2 355 ms. To compare, to build a parallel circuit with the same number

of gates, Yao's protocol takes 730 to 870 ms and the GMW protocol takes 900 to 1 200 ms. To build a sequential circuit of the same size containing AND gates, the GMW protocol may even take twice as long.

Attempting to achieve a fair comparison, we compute a hypothetical run-time for the two-party computation phase of SP1 by adding the time in column "Δ" of Table 5.32 to the time required by SP2 for the two-party computation phase. The computed time therefore includes the time required to transmit SP1's private input over the network and to send it into the enclave but excludes the time to read the circuit and to load it into the enclave. The results can be found in Table 5.34. We are aware that this approach is not perfect but we hope that it gives an adequate indication for the run-time of SP1's 2PC phase. We use this run-time for the comparison with Yao's protocol and the GMW protocol.

Table 5.34: Two-Party Computation Run-Time for SP1 and SP2. Average run-time for the two-party computation phase of SP1 and SP2 in the LAN and WAN network settings. The time required by SP2 was measured and the time required by SP1 was computed and does not include the time to read the circuit and to load it into the enclave.

Circuit Size	**Time** [ms]			
	LAN Setting		**WAN Setting**	
	SP2	**SP1**	**SP2**	**SP1**
10	89.049	223.723	386.889	1 113.098
100	89.010	224.247	386.507	1 115.523
1 000	93.014	228.734	383.181	1 111.251
10 000	99.561	237.041	397.022	1 125.837
100 000	162.523	264.637	448.012	1 129.784
1 000 000	2 006.275	2 103.603	2 303.766	2 997.543

Comparison. In Table 5.35, we compare the total run-times for Yao's garbled circuit protocol and the GMW protocol with the time we computed for SP1's two-party computation phase. Because the run-time of Yao's protocol and our Intel SGX-based 2PC protocol does not depend on the AND depth of the circuit, the run-times for sequential and parallel circuits do not differ substantially. We therefore specify averages for the computation of sequential and parallel circuits to simplify the comparison.

We set the limit for the end-to-end run-time to 10 minutes = 600 000 milliseconds. The table contains an upward pointing arrow (↑) if the computation was not completed during that time and thus, no timing result was obtained.

Table 5.35: Comparison of the Run-Times in the SFE Setting. Average total run-time for Yao's protocol, the GMW protocol and our Intel SGX-based 2PC protocol when computing sequential and parallel circuits of different sizes in (a) the LAN setting and (b) the WAN setting. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A). For Yao's protocol and our Intel SGX-based 2PC protocol, we specify averages for sequential and parallel circuits. We compare the total run-time for Yao's protocol and the GMW protocol to the time required by SP1 for the two-party computation phase.

Circuit Size	**Time** [ms]									
	X			**AX**			**A**			
	Yao	**GMW**		**Yao**	**GMW**		**Yao**	**GMW**		**SGX**
		Seq	**Par**		**Seq**	**Par**		**Seq**	**Par**	
10	4.3	2.2		4.2	6.5		4.2	9.5		223.7
100	4.3	2.2	2.1	4.4	33.3	6.6	4.4	63.1	9.5	224.2
1 000	4.4	2.2		4.6	373.0		5.2	786.8		228.7
10 000	5.8	3.0	3.0	9.2	3 829.5	39.2	12.5	7 256.9	72.7	237.0
100 000	17.1	12.0		47.5	35 085.0		79.7	↑		264.6
1 000 000	131.9	97.4	104.6	324.4	366 628.6	712.2	513.7	↑	1 387.4	2 103.6

(a) LAN Setting

Circuit Size	**Time** [ms]									
	X			**AX**			**A**			
	Yao	**GMW**		**Yao**	**GMW**		**Yao**	**GMW**		**SGX**
		Seq	**Par**		**Seq**	**Par**		**Seq**	**Par**	
10	321	160		320	530		322	796		1 113
100	321	159	159	319	2 929	531	320	5 583	793	1 116
1 000	323	160		324	26 973		322	53 543		1 111
10 000	325	161	162	420	266 628	3 039	424	↑	5 707	1 126
100 000	370	190		652	↑		879	↑		1 130
1 000 000	516	295	310	2 218	↑	29 070	3 865	↑	57 609	2 998

(b) WAN Setting

Yao's garbled ciruit protocol outperforms the GMW protocol unless circuits consisting only of XOR gates (X) are evaluated. The GMW protocol's performance is mainly determined by the circuit structure and the network latency. This becomes apparent when comparing the run-times for sequential and parallel circuits consisting of alternating layers of AND and XOR gates (AX) with circuit size 1 million. The run-time for the evaluation of the parallel circuit is decreased by factor ∼500× compared to the run-time for the evaluation

of the sequential circuit. Both have the same AND size. If that same parallel circuit is evaluated in the WAN setting, the run-time is increased by factor ∼40× compared to the LAN setting.

For circuits consisting only of XOR gates (X), the GMW protocol is up to 100× faster than our protocol. In the LAN setting, it also outperforms our protocol for sequential circuits with up to 100 gates and for all parallel circuits. In the WAN setting, our protocol is faster than the GMW protocol for all circuits with a circuit AND depth of at least 100. For a parallel circuit with circuit size 1 million, consisting only of AND gates (A), our protocol has a ∼19× better run-time.

Our Intel SGX-based 2PC protocol also narrowly outperforms Yao's protocol for circuits with 1 000 000 gates containing only AND gates (A) in the WAN setting. Its run-time is smaller by factor ∼1.3×. In all other cases, the total time required by Yao's protocol is shorter than the time required by SP1 in our protocol. In the LAN setting, it has a ∼4× better run-time than our protocol for the evaluation of the aforementioned circuit.
SP2 requires less time than SP1. It is faster than both parties in Yao's protocol for circuits with 10 000 and 100 000 gates in the WAN setting.

Table 5.36 contains the total communication of Yao's garbled circuit protocol, the GMW protocol and our Intel SGX-based 2PC protocol. For our protocol it includes Ethernet, TCP/IP and TLS protocol overheads.
The communication required by our protocol only depends on the number of circuit input and output values. In contrast, the required communication for Yao's protocol and the GMW protocol depends on the number of AND gates in the circuit. For the GMW protocol, it further depends on the circuit's AND depth.

Yao's garbled circuit protocol is constant-round. Its communication overhead mostly stems from the transfer of the garbled circuit. Per AND gate, $2 \times 128 = 256$ bits have to be transferred during the protocol's setup phase. The GMW protocol requires substantially more communication per AND gate. Its performance is highly dependent on the network latency. During the setup phase, 134 bits are transferred per multiplication triple. The multiplication triples for all AND gates of the same layer are transferred together. During the online phase, the GMW protocol requires interaction for the secure evaluation of AND gates. Two independent 2-bit messages have to be transmitted per layer of AND gates. The number of communication rounds during the online phase therefore depends on the AND depth of the circuit.

In our protocol, the circuit does not have to be transferred. This leads to the communication being constant for sequential circuits of all gate types. For parallel circuits it increases with the circuit size as parallel circuits with a larger size also have a larger number of circuit outputs. For circuits containing AND gates (AX, A), the communication overhead of our protocol is substantially smaller than that of Yao's protocol or that of the GMW protocol, both of which increase linearly with the number of AND gates. For the evaluation of a Boolean circuit with AND size 1 million and AND depth 1000, the communication of our implementation is lower by factors 12 669× and 12 976× compared to Yao's garbled circuit protocol and the GMW

protocol, respectively. For circuits consisting only of XOR gates, the GMW protocol requires less communication than our protocol.

While we were hoping for better run-time results, we remain optimistic that our proof-of-concept protocol implementation could be improved to yield better performance. Unfortunately, much of its computational overhead stems from the inherent limitations of Intel SGX such as the EPC memory size-constraints. For a sequential circuit with 1 million AND gates, we measured an average in-enclave evaluation time of 1 937 ms. Outside the enclave, using the same program, the evaluation time was by factor ∼3× smaller, averaging only around 570 ms.
Finally, we also want to note that Yao's garbled circuit protocol and the GMW protocol only provide security in the semi-honest model. Maliciously-secure cryptographic 2PC protocols have substantially longer run-times. In contrast, Intel SGX provides strong security guarantees, even against privileged attackers. This does however require trusting Intel.

Table 5.36: Comparison of the Total Communication in the SFE Setting. Total amount of data transmitted between the parties in Yao's protocol, the GMW protocol and our Intel SGX-based 2PC protocol for the computation of (a) sequential circuits and (b) parallel circuits of different sizes. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A).

Circuit Size	**Communication** [bytes]						
	X		**AX**		**A**		
	Yao	**GMW**	**Yao**	**GMW**	**Yao**	**GMW**	**SGX**
10	2 205	58	2 365	4 350	2 525	4 460	2 276
100	2 205	58	3 805	17 628	5 405	35 112	2 276
1 000	2 205	58	18 205	158 600	34 205	312 960	2 276
10 000	2 205	58	162 205	1 551 936	322 205	3 103 728	2 276
100 000	2 205	58	1 602 205	15 501 866	3 202 205	↑	2 276
1 000 000	2 205	58	16 002 232	155 004 916	32 002 268	↑	2 276

(a) Sequential Circuits

Circuit Size	**Communication** [bytes]						
	X		**AX**		**A**		
	Yao	**GMW**	**Yao**	**GMW**	**Yao**	**GMW**	**SGX**
100	2 207	60	3 807	4 372	5 407	8 598	2 278
10 000	2 229	82	162 229	179 796	322 229	355 328	2 300
1 000 000	2 453	306	16 002 480	16 389 590	32 002 516	32 778 838	2 526

(b) Parallel Circuits

5.5.4 PFE Setting

In the PFE setting, one party provides the private input and the other party provides the private function in form of the programming to the universal circuit. In Yao's garbled circuit protocol and the GMW protocol, this party is the server. In our protocol, it is SP1.
In this section, we compare the total run-time comprising setup time and online time with the time for the two-party computation phase, analogously to the SFE setting. Since the total run-times for the cryptographic protocols contain the transmission of the programming bits, we add the time required for sending SP1's input to the time required by SP2 for the two-party computation phase in order to achieve a fair comparison. The results can be found in Table 5.33. SP1's two-party computation phase ends after the result not ready message is received since it does not receive the computation result. The computed time for the 2PC phase therefore unnecessarily includes the time for the transmission of the result not ready message to SP1.

We compare the run-times for our Intel SGX-based 2PC protocol with the run-times of Yao's garbled circuit protocol and the GMW protocol in Table 5.37. The circuit sizes we specify are the sizes of the simulated circuits. The resulting universal circuits consist of 61, 1 943, 34 321 and 504 461 gates, respectively. The number of programming bits increases with the number of gates in the universal circuit.

Again, we set the limit for the end-to-end run-time to 10 minutes = 600 000 milliseconds. The table contains an upward pointing arrow (↑) if the computation was not completed during that time and thus, no timing result was obtained.

Similar to the SFE setting, our protocol cannot outperform Yao's protocol except when the largest universal circuit, simulating a circuit with 10 000 gates is computed in the WAN setting. In that case, our protcol is ∼1.5× faster than Yao's protocol. We could not evaluate and compare the run-times for larger universal circuits because the benchmarking hardware we used could not fulfil the memory requirements of the ABY framework for computing those circuits.

Table 5.37: Comparison of the Run-Times in the PFE Setting. Average total run-time for Yao's protocol and the GMW protocol when universal circuits that simulate circuits of different sizes are computed.

Circuit Size	**Time** [ms]					
	LAN Setting			**WAN Setting**		
	Yao	**GMW**	**SGX**	**Yao**	**GMW**	**SGX**
10	5.0	24.6	176.3	411.8	2 122.6	751.7
100	7.4	334.1	183.8	413.4	24 951.3	776.0
1 000	53.8	3 621.3	221.3	613.5	252 706.2	795.3
10 000	758.8	32 270.4	1 316.2	3 124.1	↑	2 104.8

The GMW protocol is only faster than our protocol for the evaluation of the smallest universal circuit in the LAN setting. In all other cases, our protocol outperforms the GMW protocol. For the evaluation of the universal circuit simulating a Boolean circuit with 10 000 gates, our protocol has a ∼24× better run-time in the LAN setting. Yao's protocol also yields a better performance than the GMW protocol for all circuit sizes.
Again, the large influence of the network latency on the performance of the GMW protocol becomes obvious when comparing the results for the LAN and WAN network settings.

Table 5.30 contains the communication for the two-party computation phase in the PFE setting along with the total communication for Yao's protocol and the GMW protocol. For the computation of the universal circuit with the largest size, the communication of our Intel SGX-based PFE solution is lower by factors ∼105× and ∼130× compared to Yao's garbled circuit protocol and the GMW protocol.

Table 5.38: Comparison of the Total Communication in the PFE Setting. Total amount of data transmitted between the parties in Yao's protocol, the GMW protocol and our Intel SGX-based 2PC protocol for the computation of universal circuits of different sizes.

Circuit Size	**Communication** [bytes]		
	Yao	**GMW**	**SGX**
10	5 254	13 209	1 591
100	97 039	199 434	2 534
1 000	1 665 574	2 496 260	18 754
10 000	24 376 321	30 250 312	232 068

6 Conclusion

Summary. Cryptographic two-party computation protocols such as Yao's garbled circuit protocol or the GMW protocol incur high computational and communication overheads. In this work, we therefore explored an alternative Intel SGX-based 2PC approach. For protection against side-channel attacks, we evaluate a Boolean circuit-representation of the function to be computed inside the enclave instead of computing the function in plain. To this effect, we designed an Intel SGX-based 2PC protocol, for which we provided a proof-of-concept implementation. In our protocol, the circuit is loaded into the enclave after its creation, making our approach applicable for general secure two-party computation. The same enclave can be reused for computing many different functions in diverse application scenarios and no costly per-application redesign is required.

Our work targeted secure function evaluation as well as universal circuit-based private function evaluation. To the best of our knowledge, we are the first to implement an Intel SGX-based PFE solution. For both the SFE and the PFE setting, we compared our implementation to the ABY framework's implementations of Yao's garbled circuit protocol and the GMW protocol in terms of performance, scalability and communication overhead. For the evaluation of a Boolean circuit with AND size 1 million and AND depth 1000, the communication of our implementation is four orders of magnitude lower compared to Yao's garbled circuit protocol and the GMW protocol. In a high-latency network setting, the run-time for our protocol's 2PC phase is smaller by factors 1.3× and 19×, respectively. For the evaluation of a universal circuit with ∼0.5 million gates, simulating a Boolean circuit with 10 000 gates, the communication of our Intel SGX-based PFE solution is lower by factors 105× and 130× compared to Yao's garbled circuit protocol and the GMW protocol. Even in a low-latency network setting, our implementation has a 24× faster run-time than the GMW protocol. In a high-latency network setting, it has a 1.5× faster run-time than Yao's protocol. Given that one trusts Intel, our Intel SGX-based approach additionally offers a stronger security model.

Limitations & Future Work. While our protocol outperforms Yao's garbled circuit protocol and the GMW protocol for circuits with large AND sizes and depths in high-latency networks due to its very low communication overhead, it still comes with a high computational overhead. This is partially due to the fact that our implementation was done as a first proof of concept and has not yet been extensively optimized. A large part of the computational overhead however also stems from the inherent limitations of Intel SGX such as the EPC memory size-constraints. EPC paging leads to long in-enclave computation times for large circuits.

Our work could be optimized and extended in several ways. First and foremost, we suggest loading the circuit into the enclave during a distinct set-up phase. If the service providers for example wish to compute the same function multiple times on different inputs, this step would not have to be repeated.

Furthermore, by adding support for multi-threading on the application side, both service providers could truly be run in parallel. Additionally, the possibility of multi-threading within the enclave could be explored.

To reduce EPC paging, one could try to optimize the circuits' memory requirements. The overall enclave size could be reduced if the circuit were evaluated in parts. For parallel circuits and circuits with low AND depths, the layer-wise evaluation might yield a positive performance effect. For sequential circuits and circuits with high AND depths, the grouping together of a larger number of layers probably makes more sense. In this case, the mechanism by which the two service providers are ensured that the right circuit is being evaluated inside the enclave might however need to be updated.

Last but not least, the security of our Intel SGX-based 2PC solution with respect to different side-channel attacks should be further investigated. While we provide an extensive overview of side-channel vulnerabilities of Intel SGX and try our best to mitigate them, a thorough side-channel analysis should be performed to ensure security. The challenge in this context seems to be keeping up with current research and maintaining security even as new attacks are being discovered.

List of Figures

List of Tables

List of Protocols

List of Abbreviations

2PC	Secure Two-Party Computation
ACK	Acknowledgement
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
AESM	Application Enclave Service Manager
API	Application Programming Interface
ASLR	Address Space Layout Randomization
BIOS	Basic Input/Output System
CMAC	Cipher-Based Message Authentication Code
CPU	Central Processing Unit
DH	Diffie-Hellman
DHKE	Diffie-Hellman Key Exchange
DRAM	Dynamic Random Access Memory
ECALL	Enclave Call
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EDL	Enclave Definition Language
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
EPID	Enhanced Privacy ID
FFI	Foreign Function Interface
GCM	Galois/Counter Mode
GID	Group ID
GMW	Goldreich-Micali-Wigderson

HTM	Hardware Transactional Memory
IAS	Intel Attestation Service
I/O	Input/Output
IP	Internet Protocol
IPP	Integrated Performance Primitives
KB	Kilobyte
KDF	Key Derivation Function
KDK	Key Derivation Key
KE	Key Exchange
L1	Level 1
L1TF	L1 Terminal Fault
L2	Level 2
L3	Level 3
LAN	Local Area Network
LLC	Last-Level Cache
LTS	Long-Term Support
MEE	Memory Encryption Engine
MK	Masking Key
MPC	Secure Multi-Party Computation
MTLS	Mutual Transport Layer Security
OCALL	Outside Call
OS	Operating System
OT	Oblivious Transfer
PFE	Private Function Evaluation
PIB	Platform Information Blob
PoET	Proof of Elapsed Time
PRM	Processor Reserved Memory
PSW	Platform Software
RA	Remote Attestation

RAM	Random Access Memory
RSA	Rivest-Shamir-Adleman
SD	Standard Deviation
SDK	Software Development Kit
SECS	SGX Enclave Control Structure
SFE	Secure Function Evaluation
SGX	Software Guard Extensions
SHA	Secure Hash Algorithm
SigRL	Signature Revocation List
SIGSTRUCT	Signature Structure
SK	Symmetric Key
SMK	Session MAC Key
SMT	Simultaneous Multithreading
SP	Service Provider
SPID	Service Provider ID
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TRE	Trustworthy Remote Entity
TSX	Transactional Synchronization Extension
UC	Universal Circuit
URL	Uniform Resource Locator
VK	Verification Key
VM	Virtual Machine
VMM	Virtual Machine Manager
WAN	Wide Area Network
XML	Extensible Markup Language

Bibliography

[AGJS13] I. ANATI, S. GUERON, S. P. JOHNSON, V. R. SCARLATA. **"Innovative Technology for CPU Based Attestation and Sealing"**. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. 2013 (cit. on pp. 9 sqq., 15).

[ALSZ17] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More Efficient Oblivious Transfer Extensions"**. In: *Journal of Cryptology* 30.3 (07/2017), pp. 805–858 (cit. on p. 4).

[AM16] J. P. AUMASSON, L. MERINO. **"SGX Secure Enclaves in Practice: Security and Crypto Review"**. In: *Black Hat USA*. 2016 (cit. on p. 24).

[ATG+16] S. ARNAUTOV, B. TRACH, F. GREGOR, T. KNAUTH, A. MARTIN, C. PRIEBE, J. LIND, D. MUTHUKUMARAN, D. O'KEEFFE, M. L. STILLWELL, D. GOLTZSCHE, D. EYERS, R. KAPITZA, P. PIETZUCH, C. FETZER. **"SCONE: Secure Linux Containers with Intel SGX"**. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 2016, pp. 689–703 (cit. on pp. 13, 19).

[BB03] D. BRUMLEY, D. BONEH. **"Remote Timing Attacks Are Practical"**. In: *Proceedings of the 12th USENIX Security Symposium (USENIX Security'03*. USENIX Association, 2003 (cit. on p. 20).

[BBB+16] R. BAHMANI, M. BARBOSA, F. BRASSER, B. PORTELA, A.-R. SADEGHI, G. SCERRI, B. WARINSCHI. **"Secure Multiparty Computation from SGX (Full Version)"**. Cryptology ePrint Archive, Report 2016/1057. `https://eprint.iacr.org/2016/1057`. 2016 (cit. on p. 26).

[BBB+17] R. BAHMANI, M. BARBOSA, F. BRASSER, B. PORTELA, A.-R. SADEGHI, G. SCERRI, B. WARINSCHI. **"Secure Multiparty Computation from SGX"**. In: *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'17)*. Springer, 2017, pp. 477–497 (cit. on p. 26).

[BCD+17] F. BRASSER, S. CAPKUN, A. DMITRIENKO, T. FRASSETTO, K. KOSTIAINEN, U. MÜLLER, A.-R. SADEGHI. **"DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization"**. In: *CoRR* abs/1709.09917 (2017) (cit. on p. 22).

[BCF+14] J. BRINGER, H. CHABANNE, M. FAVRE, A. PATEY, T. SCHNEIDER, M. ZOHNER. **"GSHADE: Faster Privacy-Preserving Distance Computation and Biometric Identification"**. In: *Proceedings of the 2nd ACM Workshop on Information Hiding and Multimedia Security (IH&MMSEC'14)*. ACM, 2014 (cit. on p. 3).

[BCKS18] M. BRANDENBURGER, C. CACHIN, R. KAPITZA, A. SORNIOTTI. **"Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric"**. In: *CoRR* abs/1805.08541 (2018) (cit. on p. 19).

[Bea92] D. BEAVER. **"Efficient Multiparty Protocols Using Circuit Randomization"**. In: *Proceedings of the 11th Annual International Cryptology Conference, Advances in Cryptology (CRYPTO'91)*. Springer, 1992, pp. 420–432 (cit. on p. 6).

[Bea95] D. BEAVER. **"Precomputing Oblivious Transfer"**. In: *Proceedings of the 15th Annual International Cryptology Conference, Advances in Cryptology (CRYPTO'95)*. Springer, 1995, pp. 97–109 (cit. on p. 4).

[BFK+09] M. BARNI, P. FAILLA, V. KOLESNIKOV, R. LAZZERETTI, A.-R. SADEGHI, T. SCHNEIDER. **"Secure Evaluation of Private Linear Branching Programs with Medical Applications (Full Version)"**. Cryptology ePrint Archive, Report 2009/195. `https://eprint.iacr.org/2009/195`. 2009 (cit. on p. 6).

[BFR+18] F. BRASSER, T. FRASSETTO, K. RIEDHAMMER, A.-R. SADEGHI, T. SCHNEIDER, C. WEINERT. **"VoiceGuard: Secure and Private Speech Processing"**. In: *Proceedings of the 19th Annual Conference of the International Speech Communication Association (INTERSPEECH'18)*. International Speech Communication Association (ISCA), 2018, pp. 1303–1307 (cit. on p. 19).

[BGNS06] E. BRICKELL, G. GRAUNKE, M. NEVE, J. SEIFERT. **"Software mitigations to hedge AES against cache-based software side channel vulnerabilities"**. Cryptology ePrint Archive, Report 2006/052. `https://eprint.iacr.org/2006/052`. 2006 (cit. on p. 22).

[BMD+17] F. BRASSER, U. MÜLLER, A. DMITRIENKO, K. KOSTIAINEN, S. CAPKUN, A.-R. SADEGHI. **"Software Grand Exposure: SGX Cache Attacks Are Practical"**. In: *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT'17)*. USENIX Association, 2017 (cit. on p. 21).

[BPH14] A. BAUMANN, M. PEINADO, G. HUNT. **"Shielding Applications from an Untrusted Cloud with Haven"**. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 2014, pp. 267–283 (cit. on pp. 13, 19).

[BPSW07] J. BRICKELL, D. E. PORTER, V. SHMATIKOV, E. WITCHEL. **"Privacy-preserving Remote Diagnostics"**. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. ACM, 2007, pp. 498–507 (cit. on p. 6).

[BT11] B. B. BRUMLEY, N. TUVERI. **"Remote Timing Attacks Are Still Practical"**. In: *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS'11)*. Springer, 2011, pp. 355–371 (cit. on p. 20).

[BWG+16] S. BRENNER, C. WULF, D. GOLTZSCHE, N. WEICHBRODT, M. LORENZ, C. FETZER, P. PIETZUCH, R. KAPITZA. **"SecureKeeper: Confidential ZooKeeper Using Intel SGX"**. In: *Proceedings of the 17th International Middleware Conference (Middleware'16)*. ACM, 2016 (cit. on p. 19).

[CCX+18] G. CHEN, S. CHEN, Y. XIAO, Y. ZHANG, Z. LIN, T. H. LAI. **"SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution"**. In: *CoRR* abs/1802.09085 (2018) (cit. on p. 22).

[CD16] V. COSTAN, S. DEVADAS. **"Intel SGX Explained"**. Cryptology ePrint Archive, Report 2016/086. `https://eprint.iacr.org/2016/086`. 2016 (cit. on pp. 9 sq., 21).

[CDD+16] F. CHEN, M. DOW, S. DING, Y. LU, X. JIANG, H. TANG, S. WANG. **"PREMIX: PRivacy-preserving EstiMation of Individual admiXture"**. In: *Proceedings of the AMIA 2016 Annual Symposium (AMIA'16)*. AMIA, 2016, pp. 1747–1755 (cit. on p. 19).

[Cha17] M. CHANDLER (INTEL CORPORATION). **"Intel(R) Enhanced Privacy ID (EPID) Security Technology"**. 07/13/2017. URL: https://software.intel.com/en-us/articles/intel-enhanced-privacy-id-epid-security-technology (visited on 11/15/2018) (cit. on p. 15).

[CS13] S. CHECKOWAY, H. SHACHAM. **"Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface"**. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 2013, pp. 253–264 (cit. on p. 13).

[CWC+18] G. CHEN, W. WANG, T. CHEN, S. CHEN, Y. ZHANG, X. WANG, T.-H. LAI, D. LIN. **"Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races"**. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP'18)*. IEEE Computer Society, 2018, pp. 178–194 (cit. on p. 23).

[CWJ+17] F. CHEN, S. WANG, X. JIANG, S. DING, Y. LU, J. KIM, S. C. SAHINALP, C. SHIMIZU, J. C. BURNS, V. J. WRIGHT, E. PNG, M. L. HIBBERD, D. D. LLOYD, H. YANG, A. TELENTI, C. S. BLOSS, D. FOX, K. LAUTER, L. OHNO-MACHADO. **"PRINCESS: Privacy-protecting Rare disease International Network Collaboration via Encryption through Software guard extensionS"**. In: *Bioinformatics* 33.6 (2017), pp. 871–878 (cit. on p. 19).

[CZRZ17] S. CHEN, X. ZHANG, M. K. REITER, Y. ZHANG. **"Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu"**. In: *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*. ACM, 2017, pp. 7–18 (cit. on pp. 20 sq.).

[DDL+17] Y. DING, R. DUAN, L. LI, Y. CHENG, Y. ZHANG, T. CHEN, T. WEI, H. WANG. **"POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave"**. In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*. Code: https://github.com/baidu/rust-sgx-sdk. ACM, 2017, pp. 2491–2493 (cit. on p. 11).

[DSZ15] D. DEMMLER, T. SCHNEIDER, M. ZOHNER. **"ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation"**. In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*. Code: https://encrypto.de/code/ABY. Internet Society, 2015 (cit. on pp. 1, 7, 44).

[EFG+09] Z. ERKIN, M. FRANZ, J. GUAJARDO, S. KATZENBEISSER, I. LAGENDIJK, T. TOFT. **"Privacy-Preserving Face Recognition"**. In: *Proceedings of the 9th International Symposium on Privacy Enhancing Technologies (PETS'09)*. Springer, 2009, pp. 235–253 (cit. on p. 3).

[FAZ05] K. B. FRIKKEN, M. J. ATALLAH, C. ZHANG. **"Privacy-preserving credit checking"**. In: *Proceedings 6th ACM Conference on Electronic Commerce (EC'05)*. 2005, pp. 147–154 (cit. on p. 6).

[GESM17] J. GÖTZFRIED, M. ECKERT, S. SCHINZEL, T. MÜLLER. **"Cache Attacks on Intel SGX"**. In: *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, 2017 (cit. on pp. 21, 24).

[GKS17] D. GÜNTHER, Á. KISS, T. SCHNEIDER. **"More Efficient Universal Circuit Constructions"**. In: *Proceedings of the 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT'17)*. Springer, 2017, pp. 443–470 (cit. on p. 7).

[GLS+17a] D. GRUSS, J. LETTNER, F. SCHUSTER, O. OHRIMENKO, I. HALLER, M. COSTA. **"Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory"**. In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, 2017, pp. 217–233 (cit. on p. 21).

[GLS+17b] D. GRUSS, M. LIPP, M. SCHWARZ, R. FELLNER, C. MAURICE, S. MANGARD. **"KASLR is Dead: Long Live KASLR"**. In: *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS'17)*. Springer, 2017, pp. 161–176 (cit. on p. 23).

[GMF+16] D. GUPTA, B. MOOD, J. FEIGENBAUM, K. BUTLER, P. TRAYNOR. **"Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation"**. In: *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'16)*. Springer, 2016, pp. 302–318 (cit. on p. 25).

[GMW87] O. GOLDREICH, S. MICALI, A. WIGDERSON. **"How to Play ANY Mental Game"**. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC'87)*. ACM, 1987, pp. 218–229 (cit. on pp. 1, 3 sqq., 7).

[GRBG18] B. GRAS, K. RAZAVI, H. BOS, C. GIUFFRIDA. **"Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks"**. In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 2018, pp. 955–972 (cit. on p. 23).

[HCP17] M. HÄHNEL, W. CUI, M. PEINADO. **"High-Resolution Side Channels for Untrusted Operating Systems"**. In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 2017, pp. 299–312 (cit. on p. 21).

[HLP+13] M. HOEKSTRA, R. LAL, P. PAPPACHAN, V. PHEGADE, J. DEL CUVILLO. **"Using Innovative Instructions to Create Trustworthy Software Solutions"**. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM, 2013 (cit. on p. 9).

[HSVW18] T. HUNT, C. SONG, R. S. a.VITALY SHMATIKOV, E. WITCHEL. **"Chiron: Privacy-preserving Machine Learning as a Service"**. In: *CoRR* abs/1803.05961 (2018) (cit. on p. 19).

[HTCK18] D. HARNIK, E. TSFADIA, D. CHEN, R. KAT. **"Securing the Storage Data Path with SGX Enclaves"**. In: *ArXiv e-prints* (06/2018), arXiv:1806.10883 (cit. on p. 24).

[IES15] G. IRAZOQUI, T. EISENBARTH, B. SUNAR. **"S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES"**. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, 2015, pp. 591–604 (cit. on p. 20).

[IKNP03] Y. ISHAI, J. KILIAN, K. NISSIM, E. PETRANK. **"Extending Oblivious Transfers Efficiently"**. In: *Proceedings of the 23rd Annual International Cryptology Conference, Advances in Cryptology (CRYPTO'03)*. Springer, 2003, pp. 145–161 (cit. on p. 4).

[Int] INTEL CORPORATION. **"PoET 1.0 Specification – Sawtooth v1.0.5 documentation"**. URL: `https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html` (visited on 12/04/2018) (cit. on p. 19).

[Int15] INTEL CORPORATION. **"Intel(R) Software Guard Extensions (Intel(R) SGX) – Tutorial Slides for the International Symposium on Computer Architecture (ISCA'15)"**. Version 1.1. 06/2015. URL: `https://software.intel.com/sites/default/files/332680-002.pdf` (visited on 11/21/2018) (cit. on p. 20).

[Int16] INTEL CORPORATION. **"Intel(R) Software Guard Extensions Part 7: Refine the Enclave with Proxy Functions"**. 11/28/2016. URL: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-7-refining-the-enclave (visited on 11/09/2018) (cit. on p. 12).

[Int18a] INTEL CORPORATION. **"Attestation Service for Intel(R) Software Guard Extensions (Intel(R) SGX): API Documentation"**. Version 4.1. 2018. URL: https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf (visited on 11/16/2018) (cit. on pp. 16, 18, 37, 41).

[Int18b] INTEL CORPORATION. **"Code Sample: Intel(R) Software Guard Extensions Remote Attestation End-to-End Example"**. 07/04/2018. URL: https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example (visited on 11/08/2018) (cit. on pp. 16, 18).

[Int18c] INTEL CORPORATION. **"Enclave Signing Tool for Intel(R) Software Guard Extensions (Intel(R) SGX)"**. 2018. URL: https://software.intel.com/sites/default/files/managed/ae/2e/Enclave-Signing-Tool-for-Intel-SGX.pdf (visited on 11/14/2018) (cit. on p. 13).

[Int18d] INTEL CORPORATION. **"Intel(R) Protected File System Library"**. 05/07/2018. URL: https://software.intel.com/en-us/sgx-sdk-dev-reference-intel-protected-file-system-library (visited on 12/17/2018) (cit. on p. 36).

[Int18e] INTEL CORPORATION. **"Intel(R) Software Guard Extensions (Intel(R) SGX) – Developer Guide"**. Version 2.3. 09/2018. URL: https://download.01.org/intel-sgx/linux-2.3.1/docs/Intel_SGX_Developer_Guide.pdf (visited on 11/06/2018) (cit. on pp. 10 sqq., 15, 24).

[Int18f] INTEL CORPORATION. **"Intel(R) Software Guard Extensions (Intel(R) SGX) SDK for Linux* OS – Developer Reference"**. Version 2.3.1. 09/2018. URL: https://download.01.org/intel-sgx/linux-2.3.1/docs/Intel_SGX_Developer_Reference_Linux_2.3.1_Open_Source.pdf (visited on 11/06/2018) (cit. on pp. 10–13, 16, 18).

[Int18g] INTEL CORPORATION. **"Intel(R) Software Guard Extensions (Intel(R) SGX) Web-Based Training – Part 1: Introduction to Intel(R) SGX"**. 01/05/2018. URL: https://software.intel.com/en-us/documentation/intel-sgx-web-based-training/intro-to-sgx (visited on 11/09/2018) (cit. on p. 11).

[Int18h] INTEL CORPORATION. **"L1 Terminal Fault"**. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault (visited on 11/27/2018) (cit. on pp. 22 sq.).

[Int18i] INTEL CORPORATION. **"Overview on Signing and Whitelisting for Intel(R) Software Guide Extension (Intel(R) SGX) Enclaves"**. 2018. URL: https://software.intel.com/sites/default/files/managed/78/4a/overview-signing-whitelisting-intel-sgx-enclaves.pdf (visited on 11/14/2018) (cit. on p. 13).

[Int18j] INTEL CORPORATION. **"Resources and Response to Side Channel L1 Terminal Fault"**. 2018. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html (visited on 11/27/2018) (cit. on p. 23).

[JKS+18] K. JÄRVINEN, Á. KISS, T. SCHNEIDER, O. TKACHENKO, Z. YANG. **"Faster Privacy-Preserving Location Proximity Schemes"**. In: *Proceedings of the 17th International Conference on Cryptology And Network Security (CANS'18)*. Springer, 2018, pp. 3–22 (cit. on p. 3).

[JSR+16] S. JOHNSON, V. SCARLATA, C. ROZAS, E. BRICKELL, F. MCKEEN. **"Intel(R) Software Guard Extensions: EPID Provisioning and Attestation Services"**. 2016. URL: https://software.intel.com/sites/default/files/managed/57/0e/ww10-2016-sgx-provisioning-and-attestation-final.pdf (visited on 11/15/2018) (cit. on p. 15).

[KHF+19] P. KOCHER, J. HORN, A. FOGH, D. GENKIN, D. GRUSS, W. HAAS, M. HAMBURG, M. LIPP, S. MANGARD, T. PRESCHER, M. SCHWARZ, Y. YAROM. **"Spectre Attacks: Exploiting Speculative Execution"**. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP'19)*. IEEE Computer Society, 2019 (cit. on p. 22).

[KHH+17] S. KIM, J. HAN, J. HA, T. KIM, D. HAN. **"Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments"**. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 2017, pp. 145–161 (cit. on p. 19).

[KMS+16] A. KOSBA, A. MILLER, E. SHI, Z. WEN, C. PAPAMANTHOU. **"Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts"**. In: *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE Computer Society, 2016, pp. 839–858 (cit. on p. 19).

[KPM+16] K. A. KÜÇÜK, A. PAVERD, A. MARTIN, N. ASOKAN, A. SIMPSON, R. ANKELE. **"Exploring the Use of Intel SGX for Secure Many-Party Applications"**. In: *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*. ACM, 2016 (cit. on p. 25).

[KPR+15] P. KOEBERL, V. PHEGADE, A. RAJAN, T. SCHNEIDER, S. SCHULZ, M. ZHDANOVA. **"Time to Rethink: Trust Brokerage using Trusted Execution Environments"**. In: *Proceedings of the 8th International Conference on Trust and Trustworthy Computing (TRUST'15)*. Springer, 2015, pp. 181–190 (cit. on p. 25).

[KS08a] V. KOLESNIKOV, T. SCHNEIDER. **"A Practical Universal Circuit Construction and Secure Evaluation of Private Functions"**. In: *Proceedings of the 12th International Conference on Financial Cryptography and Data Security (FC'08)*. Code: https://encrypto.de/code/FairplayPF. Springer, 2008, pp. 83–97 (cit. on p. 7).

[KS08b] V. KOLESNIKOV, T. SCHNEIDER. **"Improved Garbled Circuit: Free XOR Gates and Applications"**. In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08)*. Springer, 2008, pp. 486–498 (cit. on p. 5).

[KS16] Á. KISS, T. SCHNEIDER. **"Valiant's Universal Circuit is Practical"**. In: *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT'16)*. Code: https://encrypto.de/code/UC. Springer, 2016, pp. 699–728 (cit. on pp. 7 sq., 43).

[KSH+15] S. KIM, Y. SHIN, J. HA, T. KIM, D. HAN. **"A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications"**. In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets'14)*. ACM, 2015 (cit. on p. 19).

[LEPS16] J. LIND, I. EYAL, P. R. PIETZUCH, E. G. SIRER. **"Teechain: Payment Channels Using Trusted Execution Environments"**. In: *CoRR* abs/1612.07766 (2016) (cit. on p. 19).

[Lin18] Y. LINDELL. **"The Security of Intel SGX for Key Protection and Data Privacy Applications"**. 08/16/2018. URL: https://cdn2.hubspot.net/hubfs/1761386/security-of-intelsgx-key-protection-data-privacy-apps.pdf (visited on 11/21/2018) (cit. on p. 20).

[LMS16] H. LIPMAA, P. MOHASSEL, S. SADEGHIAN. **"Valiant's Universal Circuit: Improvements, Implementation, and Applications"**. Cryptology ePrint Archive, Report 2016/017. https://eprint.iacr.org/2016/017. 2016 (cit. on p. 7).

[LP07] Y. LINDELL, B. PINKAS. **"An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries"**. In: *Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT'07)*. Springer, 2007, pp. 52–78 (cit. on p. 5).

[LP08] Y. LINDELL, B. PINKAS. **"Secure Multiparty Computation for Privacy-Preserving Data Mining"**. Cryptology ePrint Archive, Report 2008/197. https://eprint.iacr.org/2008/197. 2008 (cit. on p. 3).

[LP09] Y. LINDELL, B. PINKAS. **"A Proof of Security of Yao's Protocol for Two-Party Computation"**. In: *Journal of Cryptology* 22.2 (04/2009), pp. 161–188 (cit. on p. 5).

[LSG+18] M. LIPP, M. SCHWARZ, D. GRUSS, T. PRESCHER, W. HAAS, A. FOGH, J. HORN, S. MANGARD, P. KOCHER, D. GENKIN, Y. YAROM, M. HAMBURG. **"Meltdown: Reading Kernel Memory from User Space"**. In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 2018, pp. 973–990 (cit. on p. 22).

[MAB+13] F. MCKEEN, I. ALEXANDROVICH, A. BERENZON, C. V. ROZAS, H. SHAFI, V. SHANBHOGUE, U. R. SAVAGAONKAR. **"Innovative Instructions and Software Model for Isolated Execution"**. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM, 2013 (cit. on pp. 9–12).

[Mar18] M. MARLINSPIKE. **"Technology preview: Private contact discovery for Signal"**. 09/26/2018. URL: https://signal.org/blog/private-contact-discovery/ (visited on 12/04/2018) (cit. on p. 19).

[MBF+17] S. B. MOKHTAR, A. BOUTET, P. FELBER, M. PASIN, R. PIRES, V. SCHIAVONI. **"X-Search: Revisiting Private Web Search Using Intel SGX"**. In: *Proceedings of the 18th International Middleware Conference (Middleware'17)*. ACM, 2017, pp. 198–208 (cit. on p. 19).

[MES18] A. MOGHIMI, T. EISENBARTH, B. SUNAR. **"MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX"**. In: *Proceedings of the Cryptographers' Track at the RSA Conference on Topics in Cryptology 2018 (CT-RSA'18)*. 2018, pp. 21–44 (cit. on pp. 22, 24).

[MHWK16] M. MILUTINOVIC, W. HE, H. WU, M. KANWAL. **"Proof of Luck: An Efficient Blockchain Consensus Protocol"**. In: *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*. ACM, 2016 (cit. on p. 19).

[MIE17] A. MOGHIMI, G. IRAZOQUI, T. EISENBARTH. **"CacheZoom: How SGX Amplifies the Power of Cache Attacks"**. In: *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES'17)*. Springer, 2017, pp. 69–90 (cit. on p. 21).

[NNOB12] J. B. NIELSEN, P. S. NORDHOLT, C. ORLANDI, S. S. BURRA. **"A New Approach to Practical Active-Secure Two-Party Computation"**. In: *Proceedings of the 32nd Annual International Cryptology Conference, Advances in Cryptology (CRYPTO'12)*. Springer, 2012, pp. 681–700 (cit. on p. 6).

[NPS99] M. NAOR, B. PINKAS, R. SUMNER. **"Privacy Preserving Auctions and Mechanism Design"**. In: *Proceedings of the 1st ACM Conference on Electronic Commerce (EC'99)*. ACM, 1999, pp. 129–139 (cit. on p. 3).

[NSMS14] S. NIKSEFAT, B. SADEGHIYAN, P. MOHASSEL, S. SADEGHIAN. **"ZIDS: A Privacy-Preserving Intrusion Detection System Using Secure Two-Party Computation Protocols"**. In: *The Computer Journal* 57.4 (04/2014), pp. 494–509 (cit. on p. 6).

[OSF+16] O. OHRIMENKO, F. SCHUSTER, C. FOURNET, A. MEHTA, S. NOWOZIN, K. VASWANI, M. COSTA. **"Oblivious Multi-Party Machine Learning on Trusted Processors"**. In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, 2016, pp. 619–636 (cit. on p. 19).

[OST06] D. A. OSVIK, A. SHAMIR, E. TROMER. **"Cache Attacks and Countermeasures: The Case of AES"**. In: *Proceedings of the Cryptographers' Track at the RSA Conference on Topics in Cryptology 2006 (CT-RSA'06)*. Springer, 2006, pp. 1–20 (cit. on pp. 20 sqq.).

[OTK+18] O. OLEKSENKO, B. TRACH, R. KRAHN, M. SILBERSTEIN, C. FETZER. **"Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks"**. In: *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 2018, pp. 227–240 (cit. on pp. 13 sq., 19, 23).

[PPFF16] R. PIRES, M. PASIN, P. FELBER, C. FETZER. **"Secure Content-Based Routing Using Intel Software Guard Extensions"**. In: *Proceedings of the 17th International Middleware Conference (Middleware'16)*. ACM, 2016 (cit. on p. 19).

[Rab81] M. O. RABIN. **"How To Exchange Secrets with Oblivious Transfer"**. Technical Report TR-81, Aiken Computation Laboratory, Harvard University. 1981 (cit. on p. 4).

[RTSS09] T. RISTENPART, E. TROMER, H. SHACHAM, S. SAVAGE. **"Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds"**. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, 2009, pp. 199–212 (cit. on p. 20).

[Rus17a] M. RUSSINOVICH. **"Announcing the Confidential Consortium Blockchain Framework for enterprise blockchain networks"**. 09/14/2017. URL: `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/` (visited on 12/05/2018) (cit. on p. 19).

[Rus17b] M. RUSSINOVICH. **"Announcing the Confidential Consortium Blockchain Framework for enterprise blockchain networks"**. 08/10/2017. URL: `https://azure.microsoft.com/en-us/blog/announcing-microsoft-s-coco-framework-for-enterprise-blockchain-networks/` (visited on 12/05/2018) (cit. on p. 19).

[SCF+15] F. SCHUSTER, M. COSTA, C. FOURNET, C. GKANTSIDIS, M. PEINADO, G. MAINAR-RUIZ, M. RUSSINOVICH. **"VC3: Trustworthy Data Analytics in the Cloud Using SGX"**. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, 2015, pp. 38–54 (cit. on p. 19).

[SCNS16] S. SHINDE, Z. L. CHUA, V. NARAYANAN, P. SAXENA. **"Preventing Page Faults from Telling Your Secrets"**. In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS'16)*. ACM, 2016, pp. 317–328 (cit. on p. 20).

[ŠG14] J. ŠEDĚNKA, P. GASTI. **"Privacy-preserving Distance Computation and Proximity Testing on Earth, Done Right"**. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*. ACM, 2014, pp. 99–110 (cit. on p. 3).

[SLK+17] J. SEO, B. LEE, S. KIM, M.-W. SHIH, I. SHIN, D. HAN, T. KIM. **"SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs"**. In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. 2017 (cit. on p. 20).

[SLKP17] M.-W. SHIH, S. LEE, T. KIM, M. PEINADO. **"T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs"**. In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. Internet Society, 2017 (cit. on pp. 20 sq.).

[SLTS17] S. SHINDE, D. LE TIEN, S. TOPLE, P. SAXENA. **"Panoply: Low-TCB Linux Applications with SGX Enclaves"**. In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, 2017 (cit. on pp. 13, 19).

[SWG+17] M. SCHWARZ, S. WEISER, D. GRUSS, C. MAURICE, S. MANGARD. **"Malware Guard Extension: Using SGX to Conceal Cache Attacks"**. In: *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Springer, 2017, pp. 3–24 (cit. on p. 21).

[SZ13] T. SCHNEIDER, M. ZOHNER. **"GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits"**. In: *Proceedings of the 17th International Conference on Financial Cryptography and Data Security (FC'13)*. Springer, 2013, pp. 275–292 (cit. on pp. 4, 6).

[TPV17] C.-C. TSAI, D. E. PORTER, M. VIJ. **"Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX"**. In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. Code: `https://github.com/oscarlab/graphene`. USENIX Association, 2017, pp. 645–658 (cit. on pp. 13, 19).

[TWSH18] O. TKACHENKO, C. WEINERT, T. SCHNEIDER, K. HAMACHER. **"Large-Scale Privacy-Preserving Statistical Computations for Distributed Genome-Wide Association Studies"**. In: *Proceedings of the 13th ACM Asia Conference on Computer and Communications Security (ASIACCS'18)*. ACM, 2018, pp. 221–235 (cit. on p. 3).

[Val76] L. G. VALIANT. **"Universal Circuits (Preliminary Report)"**. In: *Proceedings of the 8th Annual ACM Symposium on Theory of Computing (STOC'76)*. ACM, 1976, pp. 196–203 (cit. on pp. 1, 7).

[VMW+18] J. VAN BULCK, M. MINKIN, O. WEISSE, D. GENKIN, B. KASIKCI, F. PIESSENS, M. SILBERSTEIN, T. F. WENISCH, Y. YAROM, R. STRACKX. **"Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution"**. In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. See also: `https://foreshadowattack.eu/`. USENIX Association, 2018, pp. 991–1008 (cit. on pp. 22, 24).

[VWK+17] J. VAN BULCK, N. WEICHBRODT, R. KAPITZA, F. PIESSENS, R. STRACKX. **"Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution"**. In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. See also: `https://github.com/jovanbulck/sgx-pte`. USENIX Association, 2017, pp. 1041–1056 (cit. on p. 20).

[WAK18] N. WEICHBRODT, P.-L. AUBLIN, R. KAPITZA. **"sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves"**. In: *Proceedings of the 19th International Middleware Conference (Middleware'18)*. ACM, 2018, pp. 201–213 (cit. on p. 42).

[WCP+17] W. WANG, G. CHEN, X. PAN, Y. ZHANG, X. WANG, V. BINDSCHAEDLER, H. TANG, C. A. GUNTER. **"Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX"**. In: *Proceedings of the 24th ACM Conference Conference on Computer and Communications Security (CCS'17)*. ACM, 2017, pp. 2421–2434 (cit. on pp. 20, 23).

[WVM+18] O. WEISSE, J. VAN BULCK, M. MINKIN, D. GENKIN, B. KASIKCI, F. PIESSENS, M. SILBERSTEIN, R. STRACKX, T. F. WENISCH, Y. YAROM. **"Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution"**. Version 1.0. 08/14/2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf` (visited on 11/16/2018) (cit. on pp. 22 sqq.).

[XCP15] Y. XU, W. CUI, M. PEINADO. **"Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems"**. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, 2015, pp. 640–656 (cit. on p. 20).

[Yao86] A. C. YAO. **"How to Generate and Exchange Secrets"**. In: *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS'86)*. IEEE Computer Society, 1986, pp. 162–167 (cit. on pp. 1, 3 sqq., 7).

[YF14] Y. YAROM, K. FALKNER. **"FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack"**. In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. USENIX Association, 2014, pp. 719–732 (cit. on p. 21).

[ZCC+16] F. ZHANG, E. CECCHETTI, K. CROMAN, A. JUELS, E. SHI. **"Town Crier: An Authenticated Data Feed for Smart Contracts"**. In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS'16)*. ACM, 2016, pp. 270–282 (cit. on p. 19).

[ZDB+17] W. ZHENG, A. DAVE, J. G. BEEKMAN, R. A. POPA, J. E. GONZALEZ, I. STOICA. **"Opaque: An Oblivious and Encrypted Distributed Analytics Platform"**. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 2017, pp. 283–298 (cit. on p. 19).

[ZJRR12] Y. ZHANG, A. JUELS, M. K. REITER, T. RISTENPART. **"Cross-VM Side Channels and Their Use to Extract Private Keys"**. In: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*. ACM, 2012, pp. 305–316 (cit. on p. 20).

A Appendix

A.1 Implementation

A.1.1 Remote Attestation Code Sample

The basic directory layout of the RA sample project can be seen in Figure A.1. Only the top-level directories and the most relevant files are shown. The sample project consists of two separate programs: the `Application`, which includes the `enclave`, and the `ServiceProvider`. Only the former needs to be run on an SGX-enabled platform. The service provider and the untrusted part of the application are written in C++ while the enclave is written in the Rust programming language.
The settings file `GeneralSettings.h` is used to configure both the application and the service provider. Both programs include a file called `isv_app.cpp`, which contains the respective main method. The protocol message flow is controlled by an instance of `MessageHandler` on the application side and an instance of `VerificationManager` on the service provider side. The latter delegates most of the work to an instance of `ServiceProvider`. The message exchange between the application and the service provider is performed with the help of Google Protocol Buffers. The message formats are defined inside the `Messages.proto` file. The `WebService` is used by the service provider for communicating with the Intel Attestation Service. Boost.Asio is used for networking and by default, the communication between the service provider and the application is secured with TLS. The application takes the role of the TLS server, requiring a certificate for authentication.

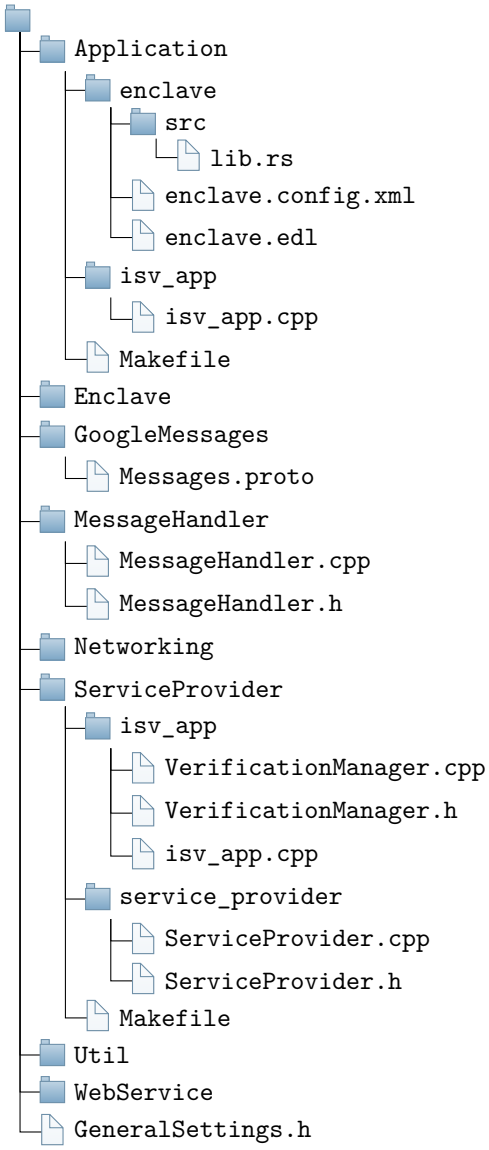


Figure A.1: Basic Directory Layout of the RA Sample Project

Both programs are built with the help of a Makefile and use some of the same utility classes, i.e., for networking. The Application Makefile automatically runs the Edger8r Tool as well as the Enclave Signing Tool (also see Section 2.4.2). As illustrated in Figure A.2, the Edger8r Tool generates the four files `enclave_t.h`, `enclave_t.c`, `enclave_u.h` and `enclave_u.c` from the EDL `enclave.edl` which defines the enclave's interface. The Enclave Signing Tool signs the enclave shared library.

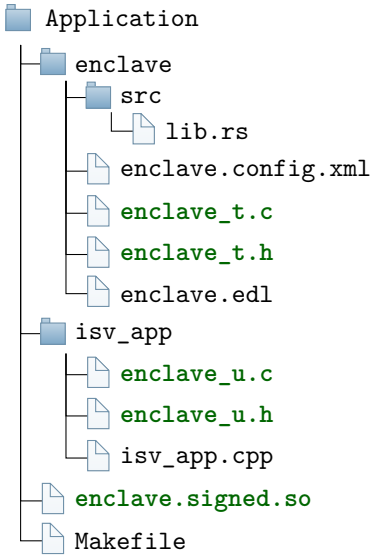


Figure A.2: Output of the Edger8r and Enclave Signing Tools

A.2 Evaluation

A.2.1 Intel SGX-Based Secure Two-Party Computation

Loading the Circuit into the Enclave. Table A.1 contains the times required for loading sequential and parallel Boolean circuits with different gate types and circuit sizes into the enclave in the SFE setting.

Table A.1: Time Required for Loading Different Circuits into the Enclave. Time needed to load (a) sequential circuits and (b) parallel circuits of different sizes into the enclave. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A). The time to parse and hash the circuit in-enclave is included. Values marked with an asterisk only cover the majority of executions but not all of them.

Circuit Size	**Time** [ms]				
	A	**X**	**AX**	**avg**	**SD**
10	0.497	0.508	0.493	0.500	0.062
100	*0.906	*0.919	*0.942	*0.923	*0.100
1 000	5.811	5.774	5.879	5.821	0.535
10 000	*19.775	*19.726	*19.745	*19.748	*0.763
100 000	205.080	203.239	204.003	204.107	6.165
1 000 000	2 271.920	2 273.780	2 295.586	2 280.429	13.246

(a) Sequential Circuits

Circuit Size	**Time** [ms]				
	A	**X**	**AX**	**avg**	**SD**
100	*0.934	*0.918	*0.865	*0.905	*0.125
10 000	*19.434	*19.620	*19.702	*19.583	*0.491
1 000 000	2 274.730	2 272.976	2 293.029	2 280.245	11.453

(b) Parallel Circuits

Evaluating the Circuit inside the Enclave. Table A.2 contains the times required for evaluating sequential and parallel Boolean circuits with different gate types and circuit sizes inside the enclave in the SFE setting.

Table A.2: Time Required for Evaluating Different Circuits inside the Enclave. Time needed to evaluate (a) sequential circuits and (b) parallel circuits of different sizes inside the enclave. The circuits either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A). The time to check the three circuit hash values for equality and to encrypt the computation result and marshal it out of the enclave is included. Values marked with an asterisk only cover the majority of executions but not all of them.

Circuit Size	**Time** [ms]				
	A	**X**	**AX**	**avg**	**SD**
10	0.172	0.173	0.174	0.173	0.012
100	0.348	0.341	0.332	0.340	0.023
1 000	2.245	2.208	2.213	2.222	0.040
10 000	*9.349	*9.550	*9.488	*9.462	*0.522
100 000	*59.866	*59.142	*58.382	*59.163	*2.650
1 000 000	1 901.876	1 901.890	1 899.891	1 901.219	6.598

(a) Parallel Circuits

Circuit Size	**Time** [ms]				
	A	**X**	**AX**	**avg**	**SD**
100	0.346	0.331	0.342	0.339	0.033
10 000	9.687	*9.705	*9.416	*9.613	*0.444
1 000 000	1 936.503	1 931.509	1 932.069	1 933.360	10.291

(b) Sequential Circuits

Starting Both Service Providers at the Same Time. The benchmarking results in Section 5.3.2 are for starting the second service provider (SP2) 6 seconds after the first (SP1). The results in this section are for starting both service providers at the same time. Due to the application and enclave being single-threaded, this leads to long run-times, which are equal for both service providers. It also leads to large standard deviations for the remote attestation phase and the two-party computation phase.
The time required for the two-party computation phase as well as the required total time, which additionally includes the remote attestation phase, are summarized in Table A.3. We report the timings for sequential circuits and select circuit sizes, which suffice to convey the idea.

Table A.3: Two-Party Computation Run-Time in the PFE Setting. Average run-time for the two-party computation phase and average total time for computing circuits of different sizes in (a) the LAN setting and (b) the WAN setting.

	2PC Phase		**Total Time**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	151.691	89.519	1 500.564	309.811
10 000	159.946	58.672	1 474.400	134.873
1 000 000	4 182.935	789.700	5 801.592	172.112

(a) LAN Setting

	2PC Phase		**Total Time**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	577.705	188.984	3 173.953	191.523
10 000	608.276	191.428	3 167.298	177.840
1 000 000	4 691.927	51.119	7 280.139	98.391

(b) WAN Setting

A.2.2 ABY-Based Secure Two-Party Computation

Setup Phase. Table A.4 contains the average run-time results for the setup phase of Yao's protocol along with the corresponding standard deviations. We separately specify the run-times for the server and the client as reported by the ABY framework.

Table A.4: Setup Phase Run-Time for Yao's Protocol in the SFE Setting. Average run-time for the setup phase when computing (a) sequential circuits in the LAN setting, (b) parallel circuits in the LAN setting, (c) sequential circuits in the WAN setting and (d) parallel circuits in the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A). For each circuit size, the first row of values is for the server and the second row of values is for the client.

	X		**AX**		**A**	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	0.791	0.685	1.071	0.645	0.819	0.827
	1.637	0.103	1.661	0.071	1.696	0.142
100	0.672	0.633	0.717	0.628	1.029	0.844
	1.711	0.064	1.756	0.181	1.762	0.129
1 000	0.821	0.523	1.000	0.643	1.644	0.697
	1.707	0.064	1.976	0.095	2.307	0.143
10 000	1.770	0.892	2.788	0.735	3.660	0.680
	2.339	0.181	4.973	0.176	7.717	0.275
100 000	5.545	0.471	16.161	1.269	28.632	2.375
	6.784	0.322	32.801	1.267	60.384	2.261
1 000 000	50.663	2.813	165.530	7.104	288.126	11.230
	51.350	2.657	205.208	6.644	342.664	9.953

(a) Sequential Circuits + LAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	1.010	0.781	1.103	0.764	0.805	0.646
	1.738	0.123	1.752	0.092	1.759	0.100
10 000	1.619	0.836	3.245	0.675	3.587	0.583
	2.298	0.198	4.978	0.174	7.652	0.266
1 000 000	52.122	3.432	167.640	4.525	296.899	18.813
	52.632	3.662	207.551	4.325	349.783	17.516

(b) Parallel Circuits + LAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	38.526	12.251	37.574	14.932	31.779	16.891
	107.255	2.901	109.304	2.252	110.240	2.173
100	33.527	17.147	29.530	18.758	28.736	17.029
	108.281	2.445	108.637	2.283	109.727	1.651
1 000	32.821	16.966	36.883	15.773	31.923	16.093
	108.450	3.580	110.119	1.851	110.178	1.494
10 000	44.105	4.730	37.964	19.788	49.145	10.335
	111.512	4.460	203.177	6.010	205.900	5.291
100 000	64.185	16.676	95.107	34.953	149.672	29.499
	128.175	3.022	419.272	23.383	648.742	43.332
1 000 000	129.192	40.080	239.981	50.053	331.686	35.699
	199.807	32.377	1 871.232	72.777	3 482.470	76.052

(c) Sequential Circuits + WAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	331.709	14.546	19.325	16.059	32.712	21.3641
	108.639	2.979	108.916	2.682	109.665	2.944
10 000	46.068	4.558	47.927	8.508	53.957	12.0261
	108.553	2.775	203.102	4.066	204.123	5.933
1 000 000	156.041	45.305	249.855	40.337	365.517	25.8141
	221.747	31.732	1 885.767	125.435	3 472.997	117.171

(d) Parallel Circuits + WAN Setting

Online Phase. Table A.5 contains the average run-time results for the online phase of Yao's protocol along with the corresponding standard deviations. We separately specify the run-times for the server and the client as reported by the ABY framework.

Table A.5: Online Phase Run-Time for Yao's Protocol in the SFE Setting. Average run-time for the online phase when computing (a) sequential circuits in the LAN setting, (b) parallel circuits in the LAN setting, (c) sequential circuits in the WAN setting and (d) parallel circuits in the WAN setting. The circuits are of different sizes and either consist only of XOR gates (X) or consist of alternating layers of AND gates and XOR gates (AX) or consist only of AND gates (A). For each circuit size, the first row of values is for the server and the second row of values is for the client.

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	2.595	0.140	2.553	0.175	2.579	0.226
	2.652	0.267	2.583	0.199	2.548	0.237
100	2.598	0.192	2.652	0.220	2.478	0.116
	2.617	0.261	2.687	0.159	2.584	0.184
1 000	2.633	0.124	2.758	0.123	3.152	0.198
	2.677	0.125	2.656	0.149	2.876	0.165
10 000	3.353	0.193	5.508	0.323	7.840	0.226
	3.428	0.193	3.963	0.262	4.613	0.119
100 000	10.404	0.416	30.552	1.827	50.438	2.046
	10.327	0.405	14.670	1.117	19.360	2.175
1 000 000	79.550	2.884	157.029	3.177	217.533	2.675
	79.532	2.881	117.649	2.878	163.657	3.680

(a) Sequential Circuits + LAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	2.591	0.092	2.497	0.153	2.592	0.147
	2.624	0.157	2.580	0.253	2.619	0.240
10 000	3.358	0.171	5.984	1.187	8.173	1.334
	3.445	0.166	4.539	1.064	5.044	1.209
1 000 000	80.376	2.437	157.323	2.286	223.615	10.155
	80.360	2.411	118.346	1.836	171.337	10.260

(b) Parallel Circuits + LAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
10	214.137	4.223	214.610	2.492	215.714	3.355
	213.836	5.131	210.284	3.551	212.144	2.680
100	214.906	6.653	213.467	4.201	213.819	3.547
	212.250	4.656	210.180	3.193	208.809	3.386
1 000	215.670	3.091	217.790	3.261	215.574	2.019
	214.182	4.847	214.356	2.782	211.817	3.844
10 000	218.478	4.398	310.562	7.717	310.033	6.476
	213.615	4.237	217.766	5.476	218.848	5.824
100 000	246.666	5.258	488.762	29.139	670.242	40.266
	242.127	4.870	232.797	11.023	229.768	5.389
1 000 000	307.065	11.597	1 904.270	61.384	3 466.422	82.038
	304.553	10.371	338.923	14.430	390.611	21.203

(c) Sequential Circuits + WAN Setting

	X		AX		A	
Circuit Size	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]	**Time** [ms]	**SD** [ms]
100	214.720	2.943	214.169	2.302	214.403	4.151
	211.892	3.559	211.147	2.140	211.081	4.531
10 000	217.302	3.307	307.313	4.278	306.115	4.889
	215.332	4.432	215.740	3.946	219.915	5.717
1 000 000	306.212	3.864	1 914.423	99.753	3 425.109	125.153
	305.056	4.837	340.471	13.789	381.711	13.609

(d) Parallel Circuits + WAN Setting