Secure and Private Function Evaluation with Intel SGX

Susanne Felsen
TU Darmstadt, Germany
susanne.felsen@gmx.de

Ágnes Kiss
TU Darmstadt, Germany
kiss@encrypto.cs.tu-darmstadt.de

Thomas Schneider
TU Darmstadt, Germany
schneider@encrypto.cs.tu-darmstadt.de

Christian Weinert
TU Darmstadt, Germany
weinert@encrypto.cs.tu-darmstadt.de

ABSTRACT

Secure function evaluation (SFE) allows two parties to jointly evaluate a publicly known function without revealing their respective inputs. SFE can be realized via well-known cryptographic protocols, such as Yao's garbled circuits (GC) and the protocol of Goldreich, Micali, and Wigderson (GMW), which obliviously evaluate a Boolean circuit representation of the function. Unfortunately, even with the most recent optimizations that touch known lower bounds, these protocols incur an impractical high communication overhead.

In this work, we efficiently realize SFE by evaluating the function in a trusted execution environment (TEE), concretely the widely available Intel Software Guard Extensions (SGX). We address the unresolved issue of countless software side-channel vulnerabilities in a unique way, namely by evaluating Boolean circuits – as used by cryptographic SFE protocols – inside an Intel SGX enclave. This way, all possible paths of the function are executed and all memory accesses are guaranteed to be independent of the actual input data.

The communication of our protocol depends only on the number of inputs and outputs (it is optimal up to an additive constant), but not on the circuit size (in contrast to Yao's GC and the GMW protocol). Furthermore, we are the first to also address efficient private function evaluation (PFE) via TEEs, where one of the parties provides a function that represents intellectual property and is computed obliviously on the other party's input. For realizing PFE, we securely evaluate universal circuits (UCs) that can be programmed via input bits to emulate any function up to a given size.

We provide a prototype implementation of our SFE and PFE protocols based on Intel SGX. We empirically compare its performance to the ABY framework (Demmler et al., NDSS'15) that provides state-of-the-art implementations of Yao's GC and the GMW protocol as well as an adapter for evaluating UCs. For PFE with a UC emulating a circuit with 10 000 gates (representing, e.g., a secret function to calculate individual car insurance rates), we improve the run-time by factor 2.3x over Yao's GC and by at least two orders of magnitude over the GMW protocol in a high-latency Internet setting. The communication is reduced by factors 106x and 131x compared to Yao's GC and the GMW protocol, respectively.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; *Privacy protections*; *Software security engineering*.

KEYWORDS

Secure Computation; Secure Function Evaluation; Private Function Evaluation; Universal Circuits; Outsourcing; Intel SGX

ACM Reference Format:
Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. 2019. Secure and Private Function Evaluation with Intel SGX. In *2019 Cloud Computing Security Workshop (CCSW '19), November 11, 2019, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3338466.3358919

1 INTRODUCTION

Secure function evaluation (SFE) allows two mutually distrusting parties to jointly compute a publicly known function without revealing their respective inputs. A classic problem to motivate SFE is the so-called millionaires' problem [127], where two millionaires want to determine who is richer without revealing their net worth to each other. SFE hereby constitutes an intriguing alternative to relying on a trusted third party.

The applications of SFE are manifold and continuously growing. Exciting new application areas include, for example, privacy-preserving machine learning inference. Here, researchers suggest to evaluate all or at least some parts of a machine learning model, e.g., a *deep/convolutional neural network* (DNN/CNN) securely using SFE technology [74, 77, 89, 103, 104, 106]. This way, in a *machine learning as a service* (MLaaS) scenario, a client is able to use an online classification service such that neither the service provider learns the private input data nor the client learns the service provider's intellectual property (i.e., the model parameters). Another recently studied SFE application is the distributed analysis of extremely sensitive genomic data [7, 16, 33, 34, 39, 109, 110, 118].

SFE can be realized via well-known cryptographic protocols, most prominently Yao's garbled circuit (GC) [127] and the protocol of Goldreich, Micali, and Wigderson (GMW) [47]. Both protocols obliviously evaluate a Boolean circuit description of the desired function. However, even with the most recent optimizations, they incur a high computation and communication overhead, which often prevents practical deployment. For example, accurately classifying a single 32x32 gray-scale image from the MNIST data set using Yao's GC results in a total communication of 791 MB [106]. Moreover, due to known lower bounds, there is little hope for substantial further improvement. For example, it is necessary to transfer at least two ciphertexts per AND gate in Yao's GC protocol [129].

Sometimes not only the inputs of a function but also the function itself should be kept secret, e.g., when executing proprietary software on private data. For these cases, *private function evaluation* (PFE) allows two mutually distrusting parties to jointly evaluate a private function provided by one of the parties on private data provided by the other party while ensuring that neither of them learns anything about the other party's input. Applications include privacy-preserving credit worthiness checking [45], remote diagnostics [23], medical diagnostics [11], and intrusion detection [99]. Conceivable future applications of PFE include privacy-preserving billing as might be used in the context of smart cars or smart homes when computing individual insurance or electricity rates.

With the help of *universal circuits* (UCs), the problem of PFE can be reduced to the problem of SFE [1]. A UC is a special type of Boolean circuit that can be programmed to simulate any Boolean function up to a given size [120]. In the PFE setting, one party provides the private input and the other party provides the private function in form of programming bits to the universal circuit, while the UC itself is the publicly known function in the SFE setting. Naturally, the previously discussed limitations of interactive SFE protocols apply to the PFE setting as well, even more so since the size of the universal circuit is super-linear in the size of the simulated circuit[1].

Trusted execution environments (TEEs) like the ubiquitously available Intel *Software Guard Extensions* (SGX) represent an attractive alternative to communication-intensive cryptographic SFE protocols. Intel SGX allows applications to create so-called *enclaves*, which are isolated from all other software running on the same machine, even privileged software such as the operating system (OS), the virtual machine manager (VMM), device drivers, or the system BIOS. In an SFE scenario, the two parties can both send their private inputs to such an enclave, inside which the function can then be computed securely and from which they can afterwards retrieve the result. For this purpose, both parties only need to establish a secure channel with the enclave and make use of a *remote attestation* (RA) protocol to verify the enclave's integrity before provisioning it with their private inputs. This concept has been used, for example, for efficient privacy-preserving speech recognition [19] and for efficient privacy-preserving machine learning inference [56].

However, Intel SGX itself does not provide protection against software side-channel attacks. Instead, developers are responsible for building enclaves that are protected against side-channel adversaries who may gather power statistics, cache miss statistics, branch statistics via timing, or page access statistics via page tables [63]. This is especially difficult since new side-channel vulnerabilities of Intel SGX are continuously being discovered by the research community (e.g., [20, 48, 54, 94, 95, 112, 122, 123]). These attacks can be used to compromise the confidentiality of SGX-protected data, which represents a major obstacle when using Intel SGX as a drop-in replacement for cryptographic SFE protocols.

So far, the only option for a developer to remove this obstacle is to rely on the many mitigation strategies proposed against various types of side-channel attacks (e.g., [18, 32, 50, 51, 113–115]). The main problem however is that these defenses are usually proposed in response to a newly discovered attack, only targeting that specific attack, and failing to consider the bigger picture [123]. Many defenses are therefore unable to offer protection against alternative attack strategies or even variations of the same attack. For a comprehensive overview of various side-channel attacks against Intel SGX and corresponding mitigation techniques we refer the reader to App. A.

Our Contributions. We observe that Boolean circuits as evaluated by cryptographic SFE protocols are inherently secure against known software side-channel attacks. This is because in a Boolean circuit representation, all possible paths of the function are executed and all memory accesses are guaranteed to be independent of the actual input data. Therefore, we realize efficient SFE based on the ubiquitously available TEE implementation Intel SGX and mitigate software side-channel attacks by evaluating Boolean circuits inside an Intel SGX enclave. This enclave can be run either on the machine of one of the protocol participants or on a machine hosted by a third party like a cloud service provider. In contrast to Yao's GC and the GMW protocol, the communication complexity of our protocol is independent of the size of the circuit, but depends only on the number of inputs and outputs and is optimal up to an additive constant[2]. This property makes our protocol especially suitable for an IoT setting where bandwidth and computation power of at least one protocol participant is limited.

While many previous works suggested using Intel SGX for privacy protection, most of them focus on a specific application (e.g., [19, 28, 29, 56, 68, 91, 111]). Due to the fact that we load the circuit into the enclave during the protocol execution, we make it possible to re-use the same enclave for the secure evaluation of arbitrary functions, thereby eliminating the need for a per-application design. Besides our simple yet effective side-channel mitigation technique, this is what distinguishes our work from previous proposals to utilize Intel SGX for secure computation [10, 53, 84] that suggested to implement the functionality in SGX and left protection against side-channel attacks to the developers. For obtaining the required Boolean circuits, users can either rely on hardware synthesis tools [38, 117] or employ specialized compilers that allow to translate a high-level programming language like ANSI C into a circuit description [26, 27, 60]. This makes our approach easy to use, even by non-experts.

Furthermore, to the best of our knowledge, we are the first to address efficient private function evaluation (PFE) via TEEs, where one of the parties provides a private function that is to be computed obliviously on the other party's input. Similarly to the only available implementations of PFE [3, 52, 78, 81], we evaluate UCs that can be programmed via input bits to emulate any function up to a given size using SFE. Concretely, we evaluate Valiant's asymptotically size-optimal UCs [3, 52, 78, 87, 120] inside an Intel SGX enclave.

We provide a fully functional prototype implementation putting our protocol designs into practice. The implementation includes a circuit evaluator for Boolean and universal circuits written in Rust using the Rust SGX SDK [42]. To actually benefit from the theoretical side-channel resistance of Boolean circuits, we make

[1]Note that there are linear-complexity PFE protocols [75, 97], but they rely on expensive public-key cryptography, namely on additively homomorphic encryption.

[2]Fully homomorphic encryption (FHE) [46, 55] also achieves SFE and PFE with optimal communication complexity, but is not yet practical due to its computation complexity.

sure to evaluate all supported gate types in constant-time and independent of the input data. Since the function to be computed is publicly known (the UC in the PFE setting), we do not have to prevent leaking the gate type during circuit evaluation.

In an empirical evaluation, we compare the performance of our Intel SGX-based SFE and PFE protocols to the state-of-the-art implementations of Yao's GC and the GMW protocol as provided by the popular ABY framework [41]. To this end, we perform an extensive analysis in two realistic network settings, measuring the run-time and communication required for evaluating circuits with different structures, sizes, and gate types for SFE as well as for evaluating UCs with different sizes for PFE. Due to the optimal communication complexity of our protocols, we can report communication improvements by several orders of magnitude.

In a high-latency Internet setting, we improve the run-time by factor 3x over Yao's GC protocol for smaller circuits and by more than an order of magnitude over the GMW protocol for a circuit with 1 million gates (with equal number of AND and XOR gates) and AND depth 500 for SFE, and by factor 2.3x over Yao's GC protocol and by over two orders of magnitude over the GMW protocol for a UC generated for a simulated circuit size of 10 000 gates for PFE. For SFE, the measured communication is reduced from 15.26 MB / 15.63 MB to 2.47 kB, and for PFE from 23.25 MB / 28.85 MB to 0.22 MB compared to Yao's GC / the GMW protocol.

In short, we summarize our contributions as follows:

- Design of software side-channel resistant Intel SGX-based SFE and PFE protocols with optimal communication complexity depending only on the number of inputs and outputs.
- Application-independent prototype implementation in Rust including a circuit evaluator with constant-time gate evaluation as well as data-independent memory accesses.
- Performance comparison to state-of-the-art implementations of Yao's GC and the GMW protocol as provided by ABY [41] showcasing the advantages and practicality of our approach.

2 PRELIMINARIES

We introduce the necessary background on secure computation in §2.1 and on Intel SGX in §2.2.

2.1 Secure Computation

2.1.1 SFE. Secure function evaluation (SFE), or secure two-party computation (2PC), 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. Prominent examples of SFE protocols based on Boolean circuits are Yao's garbled circuit (GC) protocol [127] and the protocol of Goldreich, Micali, and Wigderson (GMW) [47]. In both protocols, the evaluation of XOR gates is essentially free, i.e., requires negligible computation and no communication [82]. Therefore, the number of AND gates (multiplicative size) and the maximum number of AND gates on any path from an input to an output (multiplicative depth) of the circuit are the main cost metrics in SFE.

Oblivious Transfer (OT) is a two-party protocol where a sender inputs two values (m_0, m_1) and the receiver inputs a private selection bit σ. As a result, the receiver learns m_σ but does not learn anything about $m_{1-\sigma}$ while the sender does not learn the receiver's selection bit σ. The efficiency of OT can be improved using OT pre-computation [13] and OT extension [8, 71], where so-called *base OTs* that are generated using public-key cryptography can be extended to any number of OTs using symmetric-key cryptography.

Yao's garbled circuit (GC) protocol [127] is a constant-round protocol that works as follows. One party, the *garbler*, generates the so-called *garbled circuit*, and sends it to the other party, the *evaluator*, who evaluates it in a gate-by-gate manner based on its own and the garbler's input wire keys. The evaluator receives its input wire keys via OTs, i.e., the protocol requires one OT per input bit of the evaluator. Due to state-of-the-art optimizations, two symmetric ciphertexts per AND gate [82, 129] are sufficient to represent the garbled circuit.

In the GMW protocol [47], the two parties XOR share their inputs x and y such that $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$. One party then holds x_0, y_0, the other party holds x_1, y_1. Thereafter, the parties evaluate the circuit in an interactive manner. XOR gates can be evaluated locally on the shares, but AND gates require communication. Fortunately, all AND gates of the same circuit layer can be evaluated in parallel, and therefore, the number of communication rounds in the GMW protocol depends on the multiplicative depth of the circuit, since it is evaluated layer by layer.

ABY [41][3] is a state-of-the-art framework for efficient mixed-protocol secure two-party computation. It combines secure computation schemes based on arithmetic sharing, Boolean sharing (the GMW protocol), and Yao sharing (Yao's GC protocol). ABY provides security in the semi-honest (passive) adversary model.

2.1.2 PFE. Private function evaluation (PFE) allows two mutually distrusting parties to jointly evaluate a *private* function f provided by one of the parties on private data x provided by the other party while ensuring that neither of them learns anything about the other party's input. In the PFE setting, generally only the party providing the private data receives the evaluation result $z = f(x)$ since the function holder could choose an arbitrary function (e.g., $f(x) = x$) and therefore compromise the privacy of the other party. Though there exist different approaches for PFE with $O(k)$ complexity using homomorphic encryption [75, 97], where k is the size of the simulated circuit, the most efficient protocols using symmetric-key cryptography have complexity $\Omega(k \log k)$ [52, 78, 81, 97]. Among these, the most efficient instantiation uses secure evaluation of a so-called universal circuit [52, 78].

A universal circuit (UC) is a special type of Boolean circuit that can be programmed to simulate any Boolean function up to a given size k [120]. In addition to the private input $x = (x_1, \ldots, x_u)$, 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 the universal circuit, such that it computes $z = f(x)$ for any input x. In short: $UC(x, p_f) = f(x)$. Valiant proposed two asymptotically size-optimal constructions with size $\Omega(k \log k)$ and depth $\Omega(k)$ in [120], one based on a 2-way and another based on a 4-way recursive structure. The 2-way UC was brought into practice by Kiss and Schneider [78]

[3]https://github.com/encryptogroup/ABY

and by Lipmaa et al. [87] in concurrent and independent works, and the latter was implemented by Günther et al. in [52].

Universal circuits allow reducing the problem of PFE to the problem of SFE [1]: One party provides the private data x and the other party provides the private programming bits p_f for the public UC. 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 GC protocol [127] or the GMW protocol [47] and can therefore directly benefit from all SFE protocol optimizations.

2.2 Intel SGX

Intel *software guard extensions* (SGX) is an architecture extension consisting of CPU instructions and memory access changes to add *trusted execution environment* (TEE) capabilities to all Intel Core processors from the 6th generation on [5, 59, 92]. With Intel SGX, an application is partitioned into an untrusted and a trusted part called *enclave* that is supposed to execute all security-critical code. The initial enclave content (code and data) is loaded from unprotected memory and is free for inspection and might be subject to manipulation. Once the enclave has been initialized, its content is protected from modification and disclosure. Confidential data can then be provisioned to the enclave over a secure channel. Due to a mechanism called *sealing*, the provisioned secrets can be stored persistently for future enclave executions.

The enclave's code and data as well as associated SGX control structures are stored in the *enclave page cache* (EPC). The EPC and corresponding meta data are stored inside a special range of the main memory (DRAM) called *processor reserved memory* (PRM), which cannot be accessed by the system software or by peripherals. The PRM is 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* [36, 92].

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 the outside, not even by privileged system software such as the operating system (OS), the virtual machine manacher (VMM), device drivers, or the system BIOS. The *trusted computing base* (TCB) excludes system software and only includes the CPU (hardware and firmware) and the software inside the enclave, thereby reducing the attack surface to a minimum.

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. Example applications include data processing and analytics in cloud computing [21, 111, 132], genomic data analysis [28, 29], blockchain technologies [17, 68, 83, 85, 93, 107, 130], contact discovery [91], and machine learning [19, 56, 61, 100]. Furthermore, so-called shielding systems allow to run unmodified applications inside SGX enclaves [6, 12, 101, 116, 119]. Microsoft Azure has rolled out SGX-capable servers [108], making secure cloud applications with Intel SGX a reality.

2.2.1 Remote Attestation. In an era of cloud computing, Intel SGX aims to solve the problem of *secure remote computation*. It allows a *service provider* (SP) 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 *remote attestation* (RA) to verify the enclave's integrity and authenticity.

Remote attestation relies on the ability of the SGX-enabled platform to produce a credential called a *quote* that accurately reflects the enclave and platform state. It is generated by the *quoting enclave*, which is an Intel-provided architectural enclave devoted to RA. The quote is signed by the quoting enclave using the Intel *Enhanced Privacy ID* (EPID) signature scheme. The resulting signature can only be verified by the *Intel attestation service* (IAS) [64], an online service operated by Intel.

2.2.2 Side-Channel Attacks and Mitigation Techniques. It is important to note that Intel SGX itself does not provide protection against software side-channel attacks [5, 65, 66]. Instead, developers themselves are responsible for building enclaves that are protected against side-channel adversaries who may gather power statistics, cache miss statistics, branch statistics via timing, or page access statistics via page tables [63]. This is extremely difficult, especially since new side-channel vulnerabilities of Intel SGX are continuously being discovered by the research community.

For a comprehensive overview of various side-channel attacks against Intel SGX and corresponding mitigation techniques we refer the reader to App. A. Here we point out that the underlying problem is that mitigations and defenses are usually proposed in response to a newly discovered attack, only targeting that specific attack, and failing to consider the bigger picture [123]. Many defenses are therefore unable to offer protection against alternative attack strategies or even variations of the same attack.

3 RELATED WORK

In §3.1 we review previous works utilizing Intel SGX for secure computation and in §3.2 we discuss works which accelerate secure computation using other forms of hardware tokens.

3.1 Secure Computation Based on Intel SGX

We review related works that investigate the use of TEEs such as Intel SGX for secure computation. Koeberl et al. [80] proposed 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 list assessing the security properties of a TEE-based solution, including its resistance to side-channel attacks, as one of the main challenges that need to be addressed.

Gupta et al. [53] suggest SGX-supported SFE, providing a maliciously secure protocol that allows 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 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 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 contrast, we practically implement an Intel SGX-based SFE approach and compare it against solutions based on cryptographic protocols. Additionally, we provide security with regards to software side-channel attacks. Specifically, memory side-channel and timing attacks are mitigated by evaluating a Boolean circuit representation of the function to be computed inside the enclave and thereby performing memory accesses independent of input data.

Küçük et al. [84] 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. 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, the authors of [84] propose it to serve as an architectural template for other applications. They 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 protocols.

Our work obviously differs from [84] in that it only targets two-party applications. However, extending it to the many-party case is feasible (cf. §4.3.3). What distinguishes our work from many others, including [84], is that our design allows to re-use the same enclave for many different applications. This is due to the fact that the function to be computed is not implemented inside the enclave in plain. Instead, a Boolean circuit representation of the function is loaded and evaluated in 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 and does not require a per-application redesign and side-channel mitigations.

Bahmani et al. [10] proposed an Intel SGX-based approach for secure multi-party computation. Their main contribution is the notion of *labelled attested computation*, which allows multiple parties to interact concurrently and asynchronously with the same enclave to obtain attestation guarantees. The authors compare a two-party version of their Intel SGX-based to a cryptographic solution implemented with the ABY framework. They do not specify the network settings for the evaluation but report a speed-up of up to 300x for the total run-time of their considered applications. The authors of [10] briefly address the topic of side-channel attacks against Intel SGX and provide protocol implementations for their applications that are constant-time and thereby able to resist timing attacks.

Similar to [10], our work also implements an Intel SGX-based SFE solution and compares its performance to an ABY-based solution but differs from it in that a Boolean circuit representation of the function to be computed is evaluated. This does not only provide inherent security against a wide range of software side-channel attacks but also makes it unnecessary to design a new enclave for every application, which was necessary in [10]. As a result, the approach proposed in our work is much easier to use, even by non-experts. Additionally, our work supports the secure evaluation of private functions using universal circuits. However, ideas from [10] can be used to extend our work to securely handle concurrent and asynchronous enclave interaction.

In [2], Alder et al. target the Function-as-a-Service (FaaS) paradigm, where customers can easily deploy standalone functionalities in a cloud infrastructure. To provide various security guarantees like the confidentiality of the clients' inputs and the integrity of the computation, the authors suggest to execute the deployed functionalities within Intel SGX enclaves. However, they can guarantee input privacy only for functions where the control flow and memory access patterns are input-independent. Naturally, our idea of evaluating functionalities in a circuit representation inside Intel SGX enclaves can be utilized in the work of [2] to guarantee input privacy for arbitrary functions.

The authors of [35] study an SFE-SGX hybrid approach where the function to be computed is partly evaluated with Yao's GC protocol [127] and partly executed inside an Intel SGX enclave, depending on the efficiency and the security requirements for the different parts. However, in the hybrid protocol of [35], intermediate results are leaked when switching between function partitions and therefore may allow conclusions about private inputs. In contrast, in our work we utilize the efficiency of Intel SGX for the entire function, do not leak any intermediate results, and eliminate security concerns regarding side-channel leakage of the enclave execution by evaluating the function in a circuit representation.

In [43], the authors build the *functional encryption* (FE) system "Iron" using Intel SGX. FE is a cryptographic primitive that allows the holder of a specially constructed secret key to learn the output of an associated function on encrypted data, and therefore is closely related to fully homomorphic encryption and SFE. For the implemented and evaluated functionalities, Iron is data-oblivious, i.e., the control flow and memory accesses do not depend on sensitive data, and therefore is resistant to various software side-channel attacks [43]. In our work, we achieve data-obliviousness by evaluating a Boolean circuit representation of the functionality while not requiring the developer to manually mitigate side-channel attacks, which is an error-prone task.

All reviewed existing solutions that implement secure computation entirely via Intel SGX [2, 10, 43, 84] achieve performance results that are close to plain evaluation of the functionality and thus they are orders of magnitude more efficient than cryptographic protocols. In contrast, evaluating a Boolean circuit representation of the functionality as done in our work incurs a significant computational overhead. However, this way we can provide a generic mechanism to effectively prevent various software side-channel attacks and do not require developers to introduce function-specific protection measures.

3.2 Secure Computation Aided with a Hardware Token

We review several existing works that consider utilizing a hardware token such as a smartcard to improve the computation and/or communication complexity of secure computation.

Hazay and Lindell in [58] describe a trivial solution to secure function evaluation using smartcards, where the parties can send their inputs over a secure channel to the token, which then evaluates the function f and returns the output. A fault tolerant version of this approach was presented in [44] utilizing multiple tokens. These hardware tokens have very limited resources and their performance affects the performance of the application. [62] propose to use secure external memory to circumvent memory limitations, requiring cryptographic operations in the online phase. Intel SGX may be a viable alternative for the utilized hardware tokens in the above mentioned works.

In contrast to our work, these approaches are not generic: the functionality needs to be implemented on the token every time. Moreover, the utilized hardware tokens are vulnerable to side-channel attacks since, for example, execution time and memory accesses depend on the secret input received from the parties.

In the following, we recapitulate related work that is generic and supports arbitrary functionalities. Järvinen et al. in [72, 73] design and implement a token-assisted version of Yao's GC protocol [127]. They utilize a low-cost tamper-proof token issued by one of the parties and achieve communication complexity independent of the size of the circuit. In their implementation they utilize an FPGA and note that smartcards could also be used as a hardware token. Demmler et al. in [40] develop token-aided secure computation on mobile phones based on the GMW protocol [47], offloading the main workload to a pre-computation phase by introducing a secure hardware token held by one party. The hardware token is issued by a trusted third party and provides correlated randomness to both parties. Moreover, the authors make the communication in the pre-computation phase independent of the size of the circuit. Secure outsourced multi-party computation based on linear secret sharing aided with trusted hardware on each computing server has been proposed in [90] to decrease the computation and communication complexities of the evaluated protocol.

Our work differs from these works since we utilize Intel SGX itself for securely computing the functionality expressed as a Boolean circuit, and do not rely on executing any additional cryptographic protocols within the trusted execution environment.

Recently, *virtual black box* (VBB) obfuscation was brought into practice in [98]. The authors deliver the first implementation of a VBB obfuscation scheme on an FPGA chip that can hide the evaluated program (and optionally the inputs). To achieve VBB obfuscation, the authors require hardware Oblivious RAM (ORAM), hardware scratchpad memories, instruction scheduling techniques, and context switching. In contrast, we utilize universal circuits to hide the program and rely solely on the security of the underlying protocol and widely-available hardware to achieve privacy.

4 INTEL SGX FOR SFE AND PFE

In this section, we describe our Intel SGX-based SFE and PFE protocols as well as their prototype implementation.

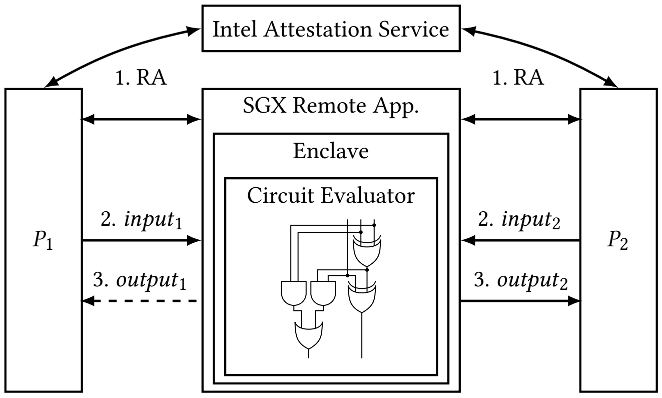


Figure 1: Overview of our Intel SGX-based SFE and PFE protocols; in the case of PFE, only P_2 receives the output.

In the basic setup as depicted in Fig. 1, we have four parties. Two parties, P_1 and P_2, correspond to the parties from cryptographic SFE and PFE protocols. They have private inputs and want to securely evaluate a (possibly private) function on these inputs. In SGX terminology, they correspond to so-called *service providers* where the service they provide is the provisioning of secrets. The third party is a so-called Intel SGX *remote application* that acts as a wrapper for the SGX enclave. This application with the enclave can be run either on the machine of one of the protocol participants or can be hosted by a third party, e.g., a cloud service provider. Finally, the *Intel attestation service* (IAS) is required in the remote attestation phase to verify the signature on the enclave quote.

In the first step of the protocol, both parties perform *remote attestation* (RA) in order to verify the enclave's integrity and authenticity before provisioning it with their secrets (cf. §2.2.1). In this phase, the parties communicate with both the SGX remote application and the IAS. This phase also includes the establishment of a secure channel between the parties and the enclave, i.e., keys are exchanged. These keys are then used for further secure communication between the parties and the enclave. Concretely, the Intel SGX SDK assists developers by providing key exchange libraries which implement a modified Sigma protocol which is used for *Diffie-Hellman key exchange* (DHKE). From the shared DH secret the parties can derive the *key derivation key* (KDK). With the help of a *key derivation function* (KDF), multiple shared keys serving different purposes are derived from the KDK.

Once trust is established and the enclave has proven that it was correctly instantiated and is running on a genuine SGX-enabled platform, the parties can send their inputs to the remote application. Both parties have private inputs to a publicly known function. In PFE, P_1's private input consists of the programming bits for a universal circuit. The parties do not transmit the circuit to the application as input during the protocol execution. Instead, the circuit files are stored on the SGX-enabled platform since the circuit (or the UC) is public and does not have to be hidden. This way, the communication complexity of our protocol does not depend on the size of the circuit, but only on the inputs and outputs and is therefore optimal up to an additive constant caused by the network protocol overhead. Note that in both cryptographic SFE protocols the communication complexity depends on the size of the circuit: In Yao's GC protocol [127], the garbled circuit is transferred from one party to the other, and in the GMW protocol [47], data is exchanged for each AND gate in the circuit (cf. §2.1).

The input messages $input_1$ and $input_2$ sent by the respective parties to the SGX remote application consist of three parts: (i) the role the party is playing in the protocol execution (i.e., P_1 or P_2), (ii) a SHA-256 hash value of the circuit *hash* that is to be used together with an AES-128 CMAC computed with a *masking key* (MK) derived from the KDK using the KDF, and (iii) the encrypted inputs. The inputs are encrypted under a *symmetric key* (SK) also derived from the KDK using the KDF. As an encryption algorithm, AES in Galois/-Counter Mode (AES-GCM) is used, which produces a ciphertext and an authentication tag, thereby providing both confidentiality and authenticity. In detail,

$$input_i = role_i || hash || CMAC_{MK_i}(hash) || AES\text{-}GCM_{SK_i}(plain_i),$$

where $plain_i$ is the party's input, i.e., either input or programming bits for SFE or PFE, respectively. The SGX SDK relies on AES-NI to implement AES-GCM, thereby preventing leaking the symmetric key via software side-channel vulnerabilities [65] (cf. App. A).

The SGX remote application is responsible for loading the specified circuit that is stored on the remote platform into the enclave. The transmitted hash value is used for identification. Since the application and the respective other party are untrusted, verifying whether the enclave contains the intended circuit before evaluating it is essential for security. The enclave code therefore checks whether the hash of the actual circuit that was loaded into the enclave matches the circuit hashes submitted by both parties and if the CMACs are valid. If this is not the case, the computation is aborted. After the circuit inside the enclave has been initialized, the SGX remote application passes the encrypted inputs to the enclave and triggers input decryption as well as circuit evaluation.

A difference between the SFE and PFE protocols is that for SFE, both parties receive the output, whereas in the PFE case only P_2 receives the output message $output_i$. This is because P_1 could choose the programming of the UC such that it compromises the privacy of P_2's input, e.g., the identity function $f(x) = x$ (cf. §2.1.2). Once the circuit evaluation is finished inside the enclave, the encrypted *result* is sent to the respective party P_i, i.e.,

$$output_i = AES\text{-}GCM_{SK_i}(result).$$

4.1 Circuit Evaluation

We briefly describe how we represent and evaluate Boolean circuits for SFE and universal circuits for PFE. We consider Boolean circuits that are composed of the functionally complete set of 2-input 1-output gates {AND, XOR}. We rely on the ABY circuit format for SFE and on the UC circuit format of [78] for PFE.

Boolean Circuits. ABY defines the ABY Boolean circuit format[4], which can be parsed by the framework [41]. It distinguishes between circuit input wires, gate output 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 gates. The wires are ordered topologically, i.e., a wire ID is always defined before it is used as a gate input or a circuit output. The ABY circuit format recognizes four gate types: XOR, AND, MUX (multiplexer), and INV (inversion).

Therefore, for securely evaluating Boolean circuits, we implement the same gate types as well for the circuit evaluation inside the Intel SGX enclave as shown in Listing 1. For side-channel protection, it is essential that the evaluation of a function gate is constant-time and does not perform any secret-dependent memory accesses (cf. §2.2). 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.

Listing 1: Evaluation of Boolean circuit gates in Rust.

```
#XOR
return inputs[0] ^ inputs[1];

#AND
return inputs[0] & inputs[1];

#MUX (cf. Kolesnikov & Schneider [82])
return ((inputs[0] ^ inputs[1]) & inputs[2]) ^
     inputs[1];

#INV
return inputs[0] ^ 0b0000_0001;
```

Although the implementation in Listing 1 fulfills these requirements, it is important to make sure that aggressive compiler optimizations do not introduce data dependent jumps. For example, in case of the MUX gate implementation, it would be reasonable to not perform the requested two XOR operations if the selection bit `inputs[2]` equals 0 but directly output `inputs[1]`. Therefore, we carefully inspected the assembler output for all gate types when compiling the code in release mode. As illustrated for the MUX gate in Listing 2, our input data-independent constant-time implementation is adopted on assembler level without any modifications, even when `inputs[2]=0`.

Listing 2: MUX gate evaluation in assembler (AT&T syntax).

```
...
movb  13(%rsp), %al   ;copy inputs[1] to a
movb  12(%rsp), %cl   ;copy inputs[0] to c
xorb  %al, %cl        ;c = c ^ a
andb  14(%rsp), %cl   ;c = c & inputs[2]
xorb  %al, %cl        ;c = c ^ a
...
```

Universal Circuits. Kiss and Schneider define a format for universal circuits[5] in [78], which can also be parsed by the ABY framework [41]. A UC in this format is generated by their UC compiler, together with the programming file corresponding to the desired functionality. Universal circuits are made up of X switches, Y switches, and universal gates. Beside the inputs, the gate output wires depend on one or more programming bits, which are read

[4]https://github.com/encryptogroup/ABY/blob/public/bin/circ/circuitformat.md

[5]https://github.com/encryptogroup/UC/blob/master/README.md

from the programming file, each line of which corresponds to a gate in the UC. X and Y switches take a single programming bit p. An X switch has two input and two output wires, and either outputs them in order (if $p = 0$) or in reverse order (if $p = 1$). A Y switch is the same as a multiplexer, it has two input wires but only one 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 and one output.

We implement the same gate types for the circuit evaluation inside the Intel SGX enclave as depicted in Listing 3 in order to privately evaluate Boolean circuits. X switches, Y switches, and universal gates are implemented using AND and XOR gates [78, 82] as in ABY to avoid secret-dependent branching and data accesses. As for the implementation of Boolean circuit gates, we carefully inspected the compilation result of the listed code on assembler level to make sure aggressive compiler optimizations do not introduce input-dependent jumps here either.

Listing 3: Evaluation of universal circuit gates in Rust.

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

#Y Switch
return ((inputs[0] ^ inputs[1]) & p) ^ inputs[1];

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

4.1.1 Protection against Side-Channel Attacks. Our implementation in Rust evaluates circuit gates without branching in constant time. Furthermore, it does not perform input data-dependent memory accesses. We also made sure that these properties are equally translated to machine code by manually inspecting the compilation result. Therefore, we effectively protect against timing and page-table- as well as cache-based software side-channel attacks (cf. §2.2.2, §A.1 and §A.2 in App. A).

However, there are speculative execution-based attacks like Foreshadow [121, 125] that do not require code vulnerabilities in the victim's enclave (cf. §2.2.2, §A.3 in App. A). Unfortunately, we currently cannot guarantee protection against these kinds of attacks and have to rely on the hardware manufacturer to incorporate mitigation techniques in next-generation CPUs via hardware changes [67, 69, 125].

4.2 Implementation Details

Our implementation is based on the Rust SGX SDK's remote attestation example[6]. It consists of two separate programs: the SGX remote application, which includes the enclave, and the SFE/PFE party. Only the former needs to be run on an SGX-enabled platform.

[6]https://github.com/baidu/rust-sgx-sdk/tree/master/samplecode/remoteattestation

The SFE/PFE party and the untrusted part of the application are written in C++ while the enclave is written in Rust. The RA example uses Google *protocol buffers*[7] (protobuf) for message exchange between the party and the application, where a series of key-value pairs is concatenated into a byte stream when the message is encoded. It therefore becomes obvious that protobuf incurs some additional communication overhead. We reduced this overhead by optimizing the encoding taken over from the RA code example, yielding smaller message sizes, and selecting the optimal encoding for newly added message types. By default, the communication between the party and the application is secured with TLS v1.2. Here, the SGX remote application takes the role of the TLS server.

4.3 Extensions

We propose two possible extensions that can further reduce the communication of our protocols when used in practice and we also explain how our protocols can be extended from the two-party to the multi-party case. However, we leave the implementation of these extensions as future work.

4.3.1 Re-Usable Inputs via Sealing. For certain applications, one of the parties might use the same set of inputs for many executions. For example, in a machine learning inference use case where one party needs to transmit the sensitive weights for an oblivious DNN/CNN evaluation [89, 104, 106], or in case of PFE where the same programming bits need to be transmitted to evaluate the same private function on different inputs. In the current protocol design, these inputs must be transferred for every single execution.

For these scenarios, we suggest to make use of the Intel SGX *sealing* feature (cf. §3.1). Sealing in this case allows the enclave to securely store the decrypted inputs persistently for future executions such that they do not need to be part of the input message. Implementing this extension requires only minor changes in the enclave code and the message format.

To illustrate the possible improvement of this extension, consider privacy-preserving speech recognition as a possible application. Here, the required neural network parameters and decoding graphs easily require the transfer of more than 500 MB [19]. Using sealing similar as in the special-purpose architecture of [19], this transfer is turned into a one-time expense.

4.3.2 On-Site Circuit Compilation. In our protocols, we assume that the public circuits already reside on the machine performing the circuit evaluation. However, they must get there beforehand somehow. Also, there might be use case scenarios where the function to be computed changes very frequently and requires a new circuit to be uploaded. Since circuits can get quite large for sophisticated functions, this would incur a high communication overhead.

For these scenarios, we suggest to compile the circuit on-site from a concise function description in a high-level programming language for which there exist various tools [26, 27, 38, 60, 117]. This requires additional steps between performing remote attestation and sending the inputs. First, one of the parties must distribute the function description to the other party and the SGX remote application. Then, each of these three entities runs the deterministic compilation process and computes the hash value of the compilation

[7]https://developers.google.com/protocol-buffers/

result. Since the compilation result is deterministic, this hash value can now be used to proceed with the protocol as described before.

To illustrate the possible improvement of this extension, consider a single double precision multiplication. This simple function can be expressed in a single line of C code which in turn can be encoded in a few bytes. In contrast, a Boolean circuit in the ABY circuit format for performing multiplications between 64-bit IEEE 754 floating point numbers has a size of 515 kB [38].

4.3.3 SFE and PFE for Multiple Parties. Not all cryptographic SFE protocols can naturally be extended to the case where more than two parties provide private inputs. The secure computation literature therefore also studies *secure multi-party computation* (SMPC) protocols (see, e.g., [14, 15, 37]). Similar to cryptographic SFE protocols, they obliviously evaluate the circuit description of a publicly known function.

Our SGX-based SFE protocol can easily be extended to an SMPC protocol that supports more than two parties. For this, instead of parties P_1 and P_2 in Fig. 1, we consider n parties, P_1 to P_n, that behave similar to P_1 and P_2, i.e., they provide private inputs over secure channels to an attested enclave and likewise receive the computation result. Generating and evaluating circuits that support input data for more than two parties presents no challenge. For a fair performance evaluation, a future implementation of this extension should be compared to cryptographic SMPC frameworks like SCALE-MAMBA [4]. In contrast to existing works that study the use of Intel SGX for SMPC protocols, we can re-use the same enclave for many different applications while providing inherent protection against a wide range of software side-channel attacks (cf. §3.1).

Extending our protocols to the multi-party case is possible not only for SFE, but also PFE. For multi-party PFE, still only one party provides the programming bits to the circuit while not receiving the function output to prevent a malicious function provider from obtaining private inputs (cf. §2.1.2). Therefore, extending our PFE protocol to the multi-party case is analogous to the above described extension for multi-party SFE. Multi-party PFE was previously considered only in [76, 96]. A possible application area could be privacy-preserving matchmaking, where protocol participants are grouped based on personal information, preferences, and statistics using criteria the matchmaker does not want to publicly disclose.

5 EVALUATION

We perform all our experiments on an SGX-enabled Intel Compute Stick STK2mv64CC equipped with an Intel Core m5-6Y57 CPU @ 1.10 GHz with 3.6 GB RAM running Ubuntu 18.04 LTS. The enclave page cache (EPC) has a maximum size of 128 MB, out of which approximately 90 MB are free to use. For evaluations with circuits consisting of more than a million gates, the maximum enclave stack and heap size had to be set to 8 kB and 205 MB, respectively.

We evaluate all protocols using two realistic network settings which are simulated using the Linux traffic control (`tc`) on the loop-back interface: (i) a LAN setting with 1 Gbit/s bandwidth and 1 ms RTT and (ii) a WAN setting with 100 Mbit/s bandwidth and 100 ms RTT. The communication between the SGX service providers and the SGX remote application as well as between the service providers and the Intel attestation service (IAS) is secured with TLS v1.2.

5.1 Benchmark Circuits

For the SFE setting, we evaluate circuits with different sizes (10 to 1 000 000 gates) and structures.

In Yao's GC and the GMW protocol evaluating XOR gates is essentially "free", whereas for AND gates communication and cryptographic operations are required (cf. §2.1). In contrast, we expect no difference in run-time for AND and XOR gates for our Intel SGX-based protocols. Since existing compilers for cryptographic SFE protocols minimize the number of AND gates and the AND depth of the circuit at the cost of many times the number of XOR gates [26, 27, 60], using such compilers for our benchmarks would present an unfair advantage for cryptographic SFE protocols. Therefore, instead of compiling circuits for example applications with such compilers, we created synthetic benchmark circuits[8].

We evaluate circuits with three different gate type compositions: The first and second consist of only AND (A) / XOR (X) gates and therefore represent the worst / best case for cryptographic SFE protocols, respectively. The third seeks to represent an average case with alternating AND and XOR gate layers (AX).

The performance of the GMW protocol is also heavily influenced by the circuit topology and the network latency since for each layer containing an AND gate one round of communication is necessary (cf. §2.1). Therefore, we benchmark "sequential" and "parallel" circuits where all of the n gates are either evaluated sequentially in n layers or $\sqrt{n}$ gates are evaluated in parallel on each of the $\sqrt{n}$ layers, respectively.

The amount of inputs highly depends on the concrete application and up to a certain point is rather independent of the circuit size and structure. That is why we keep the influence of inputs in the comparison as low as possible and feed only a single pair of input bits into all benchmark circuits. Due to their structure, all parallel circuits have $\sqrt{n}$ outputs. The construction of the different circuit structures is exemplified in Fig. 2.

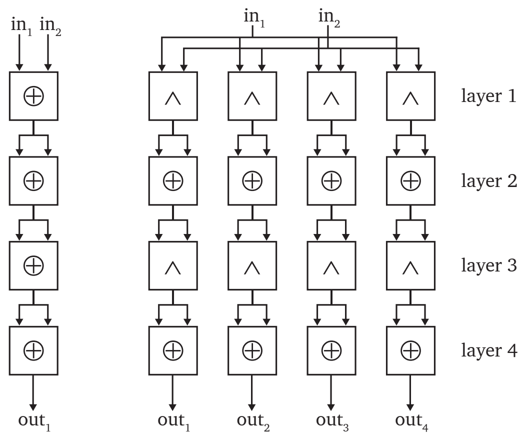


Figure 2: Example of a sequential X circuit with $n = 4$ gates (left) and a parallel AX circuit with $n = 16$ gates (right).

[8]For a fair comparison based on real applications, it would be necessary to adapt logic synthesis tools as done in [38], but with different penalties to minimize the total size of the circuit for our Intel SGX-based protocols.

Table 1: Sizes and depths of universal circuits for simulating arbitrary Boolean circuits of sizes up to $k = 10\,000$ gates.

Size of Simulated Circuit k	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
AND size	77	2 139	36 317	524 457
Total size	273	8 095	141 083	2 056 027
AND depth	38	465	4 740	47 490
Total depth	114	1 395	14 220	142 470

For the PFE setting, we generated universal circuits using the UC compiler of [3, 52, 78] to simulate arbitrary Boolean circuits of sizes up to 10 000 gates. The sizes and depths of the resulting UCs are given in Tab. 1. The AND size and depth are relevant for PFE with the evaluation of UCs using cryptographic SFE protocols (cf. §2.1), while the total size and depth are relevant for our Intel SGX-based PFE protocol. We note that our largest UC has over 2 million gates.

5.2 One-Time Expenses

Our Intel SGX-based SFE and PFE protocols as well as the ABY implementation of Yao's GC and the GMW protocol contain certain phases that count as one-time expenses. This means that such phases only need to be executed once when initializing the execution and their result can be re-used for many computations on many different functions between the same parties.

For our Intel SGX-based SFE and PFE protocols we count enclave creation and remote attestation as one-time expenses. Both parties need to individually perform RA in order to establish a secure channel with the same enclave. Once RA has been performed, the same enclave can be used for computing many different functions. The enclave can even use sealing to persistently store the secret keys that were established with the parties. The RA phase then does not have to be performed again until the enclave identity (enclave measurement) changes.

Creating the enclave takes on average 2.2 s. The RA run-time is mainly dictated by the speed of the connection with the IAS, which we did not simulate, and by the amount of data transferred, which we cannot optimize. On average, RA takes 985 ms and 1 722 ms in the LAN and WAN setting, respectively. The total communication is 64 kB for the parties and 14 kB for the SGX remote application.

For ABY, we count performing base OTs as a one-time expense. These are OTs that require costly public-key operations and are necessary to bootstrap OT extension protocols (cf. §2.1). Once an OT extension protocol is bootstrapped, it can be used to generate a virtually unlimited amount of OTs using only cheap symmetric-key operations. On average, performing base OTs took 378 ms and 562 ms in the LAN and WAN setting, respectively. The total communication for each of the two parties is 98 kB.

In summary, ABY has less one-time expenses in terms of run-time, however, for both implementations the expenses are so small that they quickly amortize over time.

5.3 Run-Time & Communication Measurements

In Fig. 3 and Fig. 4 we compare the run-times (in two network settings) and the communication of our Intel SGX-based SFE and PFE protocols, respectively, to the implementations of Yao's GC and the GMW protocol provided by the ABY framework [41]. For SFE protocols, Fig. 3 contains only the results for the AX circuits (which reflect the gate composition for real applications most accurately), for comparisons of the best and worst case scenarios of cryptographic SFE protocols, we refer the reader to Fig. 5 and Fig. 6 in App. B, respectively.

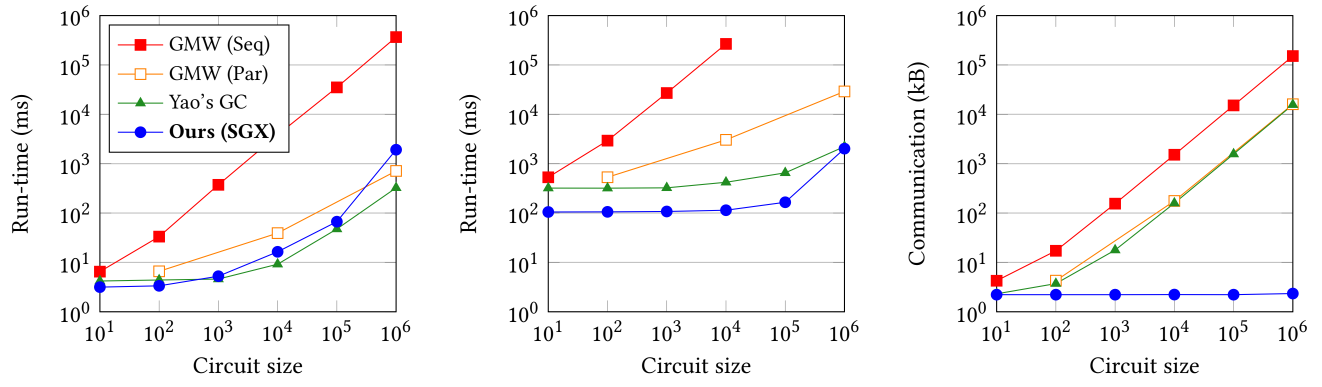
The run-times of our Intel SGX-based SFE and Yao's GC protocol for sequential and parallel circuits do not differ significantly since they do not depend on the AND depth of the circuit. For these protocols we therefore specify averages for the computation of sequential and parallel circuits to simplify the comparison and furthermore only report the slightly higher communication of parallel circuits which additionally depends on the number of output bits. All run-times are the average of 10 executions. Note that the communication results for the Intel SGX-based protocols include overhead caused by TLS v1.2 and TCP whereas ABY reports only the amount of plain data sent via a TCP socket.

The reported run-times for our Intel SGX-based implementation contain the time for transmitting the private input of one party over the network and sending it into the enclave, the circuit evaluation, and returning the result to that party. For the PFE results we report the run-time of the party transmitting the UC programming bits as its input. For a fair comparison to ABY, one-time expenses for enclave creation, remote attestation, and also the time to read the circuit and to load it into the enclave are excluded (cf. §5.2).

The measurements for ABY consist of both the so-called *setup* and the *online* phases. In the setup phase, all cryptographic operations are pre-computed except those required for evaluating garbled circuits in Yao's GC protocol. Also, a garbled version of the circuit is transferred in case of Yao's GC protocol. In the online phase, inputs are retrieved, the actual circuit evaluation takes place, and outputs are revealed. The total run-time measured by ABY by default does not include the one-time expenses for base OTs (cf. §5.2). Reading and parsing the circuit from file is also not included.

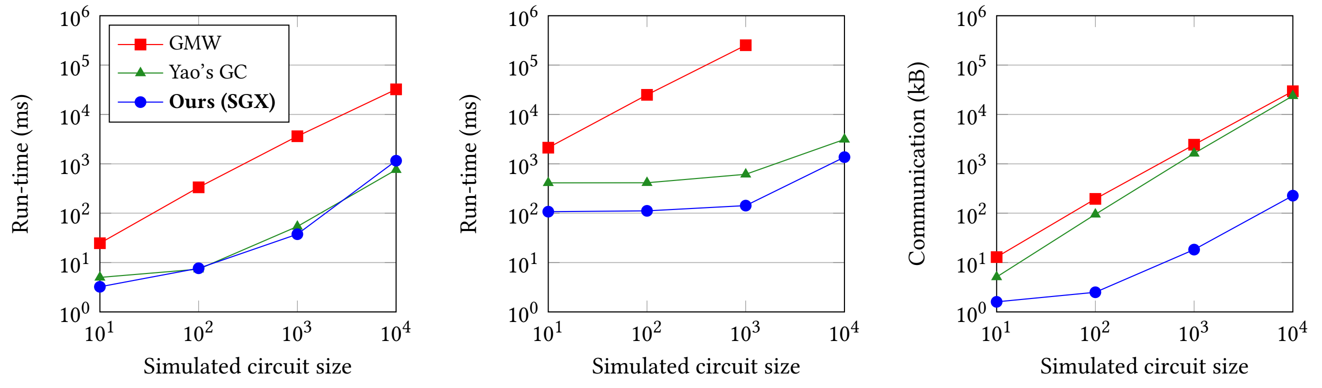
5.4 Comparison

In terms of run-time, our Intel SGX-based SFE protocol is slightly faster than Yao's GC and the GMW protocol in the LAN setting for small circuits up to 1 000 gates, and only marginally slower than Yao's GC protocol for larger circuits. However, for larger circuits that exceed EPC memory, the run-time of our protocol increases substantially. This reflects the negative performance implications of EPC paging: additional enclave transitions are required to handle page faults and decryption / encryption is necessary when swapping pages into / out of the EPC, respectively [124]. We stress that this is due to the fact that our Intel SGX circuit evaluator is a first prototype implementation, whereas the secure computation community has optimized the performance of the ABY framework over many years. With an optimized implementation that reduces the memory requirements of circuits and evaluates large circuits in



(a) Total Run-times in LAN (b) Total Run-times in WAN (c) Total Communication

Figure 3: Performance comparison of SFE protocols for AX circuits; for the Intel SGX-based and Yao's GC protocol, the differences between sequential ("seq") and parallel ("par") circuits are negligible and therefore omitted.



(a) Total Run-times in LAN (b) Total Run-times in WAN (c) Total Communication

Figure 4: Performance comparison of PFE protocols; the sizes and depths of the securely evaluated universal circuits are given in Tab. 1.

smaller parts, we expect to achieve a performance that matches or even outperforms Yao's GC protocol.

In the WAN setting, our Intel SGX-based SFE protocol is more than factor 3x faster than Yao's GC protocol up to the point where the circuit size exceeds EPC memory. This is mainly due to the improved round complexity of our protocol: when excluding remote attestation as a one-time expense, we require only one communication round that consists of sending the private inputs and receiving the computation result. In contrast, although being a constant-round protocol as well, Yao's GC protocol requires additional communication rounds for performing OT extension in the setup phase and transferring input keys via OT in the online phase. Due to its interactive online phase, the GMW protocol is one order of magnitude slower in the WAN setting for the parallel circuits and even exceeds a timeout of 10 min for large sequential circuits.

One main benefit of our Intel SGX-based SFE protocol is its optimal communication complexity that only depends on the number of inputs and outputs, but not on the size of the circuit. As a result, we can report a reduction of communication from 15.26 MB / 15.63 MB to 2.47 kB compared to Yao's GC / the GMW protocol, i.e., a reduction by three orders of magnitude for evaluating the parallel AX circuit with 1 million gates.

Similar to the SFE scenario, the run-time of our Intel SGX-based PFE protocol in the LAN setting roughly matches Yao's GC protocol and both outperform the GMW protocol. Due to the lower round complexity, the run-time in the WAN setting is reduced by a multiple of the round trip time compared to Yao's GC protocol, whereas the GMW protocol again exceeds the 10 min timeout limit for large simulated circuits. For PFE, the communication of the party providing the programming bits grows in $O(k \log k)$, where k is the size of the simulated circuit. Nevertheless, we can report a reduction of communication from 23.25 MB / 28.85 MB to only 0.22 MB compared to Yao's GC / the GMW protocol in our largest example with $k = 10\,000$, where the UC has over 2 million gates.

All in all, our Intel SGX-based protocols are an attractive alternative to cryptographic SFE protocols, especially in IoT settings where bandwidth and computation power of at least one protocol participant is limited.

6 CONCLUSIONS & FUTURE WORK

In this work, we presented the first general-purpose SFE and PFE protocols and implementations based on Intel SGX. Evaluating Boolean circuits within a secure enclave instead of the functionality itself allows us to protect against various software side-channel attacks caused by code vulnerabilities in the victim enclave. Our solution can easily be used even by non-experts by utilizing various existing compilers for translating the high-level function description to a Boolean circuit representation.

However, our experiments revealed that our circuit evaluation prototype implementation has a substantial computational overhead for larger circuits due to EPC paging caused by high memory utilization. As part of future work, we therefore plan to optimize performance by partially loading and evaluating the circuit, i.e., evaluating the circuit layer-wise (in case of parallel circuits with a low depth) or grouping together a certain number of layers (in case of sequential circuits with high depth).

While our protocols appear to be trivially secure, a proper analysis that introduces a formal notion of security and proves this notion is met by our protocol is currently lacking. Such an analysis is necessary to conduct as part of future work before deploying our protocols in practice.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within the National Research Center for Applied Cybersecurity CRISP, and by the DFG as part of project E4 within the CRC 1119 CROSSING and project A.1 within the RTG 2050 "Privacy and Trust for Mobile Users".

REFERENCES

[1] M. Abadi and J. Feigenbaum. 1990. Secure Circuit Evaluation. In *J. Cryptology*.
[2] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. 2018. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *arXiv preprint 1810.06080*.
[3] M. Y. Alhassan, D. Günther, Á. Kiss, and T. Schneider. 2019. Efficient and Scalable Universal Circuits. In *Cryptology ePrint Archive, Report 2019/348*.
[4] A. Aly, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. P. Smart, and T. Wood. 2019. SCALE–MAMBA v1.5: Documentation.
[5] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.
[6] 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, and C. Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[7] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin. 2018. Privacy-Preserving Search of Similar Patients in Genomic Data. In *PETS*.
[8] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. 2017. More Efficient Oblivious Transfer Extensions. In *J. Cryptology*.
[9] J. P. Aumasson and L. Merino. 2016. SGX Secure Enclaves in Practice: Security and Crypto Review. In *Black Hat USA*.
[10] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. 2017. Secure Multiparty Computation from SGX. In *FC*.
[11] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. 2009. Secure Evaluation of Private Linear Branching Programs with Medical Applications. In *ESORICS*.
[12] A. Baumann, M. Peinado, and G. Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[13] D. Beaver. 1995. Precomputing Oblivious Transfer. In *CRYPTO*.
[14] D. Beaver, S. Micali, and P. Rogaway. 1990. The Round Complexity of Secure Protocols (Extended Abstract). In *STOC*.
[15] M. Ben-Or, S. Goldwasser, and A. Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*.
[16] C. Bonte, E. Makri, A. Ardeshirdavani, J. Simm, Y. Moreau, and F. Vercauteren. 2018. Towards Practical Privacy-Preserving Genome-Wide Association Study. In *BMC Bioinformatics*.
[17] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti. 2018. Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric. In *arXiv preprint 1805.08541*.
[18] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A.-R. Sadeghi. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. In *arXiv preprint 1709.09917*.
[19] F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert. 2018. VoiceGuard: Secure and Private Speech Processing. In *INTERSPEECH*.
[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[21] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware*.
[22] E. Brickell, G. Graunke, M. Neve, and J. Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. In *Cryptology ePrint Archive, Report 2006/052*.
[23] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. 2007. Privacy-preserving Remote Diagnostics. In *CCS*.
[24] B. B. Brumley and N. Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *ESORICS*.
[25] D. Brumley and D. Boneh. 2003. Remote Timing Attacks Are Practical. In *USENIX Security*.
[26] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *CCS*.
[27] N. Büscher, A. Holzer, A. Weber, and S. Katzenbeisser. 2016. Compiling Low Depth Circuits for Practical Secure Computation. In *ESORICS*.
[28] F. Chen, M. Dow, S. Ding, Y. Lu, X. Jiang, H. Tang, and S. Wang. 2016. PREMIX: PRivacy-preserving EstiMation of Individual admiXture. In *American Medical Informatics Association Annual Symposium (AMIA)*.
[29] 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, and L. Ohno-Machado. 2017. PRINCESS: Privacy-protecting Rare disease International Network Collaboration via Encryption through Software guard extensionS. In *Bioinformatics*.
[30] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. In *arXiv preprint 1802.09085*.
[31] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin. 2018. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE S&P*.
[32] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ASIACCS*.
[33] K. Cheng, Y. Hou, and L. Wang. 2018. Secure Similar Sequence Query on Outsourced Genomic Data. In *ASIACCS*.
[34] H. Cho, D. J Wu, and B. Berger. 2018. Secure Genome-Wide Association Analysis using Multiparty Computation. In *Nature Biotechnology*.
[35] J. I. Choi, D. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor. 2019. A Hybrid Approach to Secure Function Evaluation Using SGX. In *ASIACCS*.
[36] V. Costan and S. Devadas. 2016. Intel SGX Explained. In *Cryptology ePrint Archive, Report 2016/086*.
[37] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*.
[38] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni. 2015. Automated Synthesis of Optimized Circuits for Secure Computation. In *CCS*.
[39] D. Demmler, K. Hamacher, T. Schneider, and S. Stammler. 2017. Privacy-Preserving Whole-Genome Variant Queries. In *CANS*.
[40] D. Demmler, T. Schneider, and M. Zohner. 2014. Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens. In *USENIX Security*.
[41] D. Demmler, T. Schneider, and M. Zohner. 2015. ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.
[42] Y. Ding, R. Duan, L. Li, Y. Cheng, Y. Zhang, T. Chen, T. Wei, and H. Wang. 2017. POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *CCS*. Code: https://github.com/baidu/rust-sgx-sdk.
[43] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. 2017. IRON: Functional Encryption using Intel SGX. In *CCS*.
[44] M. Fort, F. C. Freiling, L. D. Penso, Z. Benenson, and D. Kesdogan. 2006. TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. In *ESORICS*.

[45] K. B. Frikken, M. J. Atallah, and C. Zhang. 2005. Privacy-Preserving Credit Checking. In *ACM Conference on Electronic Commerce (EC)*.

[46] C. Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *STOC*.

[47] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *STOC*.

[48] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. 2017. Cache Attacks on Intel SGX. In *European Workshop on Systems Security (EuroSec)*.

[49] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[50] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *USENIX Security*.

[51] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. 2017. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*.

[52] D. Günther, Á. Kiss, and T. Schneider. 2017. More Efficient Universal Circuit Constructions. In *ASIACRYPT*.

[53] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor. 2016. Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation. In *FC*.

[54] M. Hähnel, W. Cui, and M. Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX ATC*.

[55] S. Halevi and V. Shoup. 2014. HElib – An Implementation of Homomorphic Encryption. (2014). https://github.com/shaih/HElib

[56] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. 2018. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. In *arXiv preprint 1808.00590*.

[57] D. Harnik, E. Tsfadia, D. Chen, and R. Kat. 2018. Securing the Storage Data Path with SGX Enclaves. In *arXiv preprint 1806.10883*.

[58] C. Hazay and Y. Lindell. 2008. Constructions of Truly Practical Secure Protocols using Standard Smartcards. In *CCS*.

[59] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[60] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. 2012. Secure Two-Party Computations in ANSI C. In *CCS*.

[61] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. In *arXiv preprint 1803.05961*.

[62] A. Iliev and S. Smith. 2005. *More Efficient Secure Function Evaluation Using Tiny Trusted Third Parties*. Technical Report TR2005-551. Dartmouth College, Hanover, NH.

[63] Intel Corporation. 2015. Intel(R) Software Guard Extensions (Intel(R) SGX) – Tutorial Slides for the International Symposium on Computer Architecture (ISCA). (2015). https://software.intel.com/sites/default/files/332680-002.pdf

[64] Intel Corporation. 2018. Attestation Service for Intel(R) Software Guard Extensions (Intel(R) SGX): API Documentation. (2018). https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf

[65] Intel Corporation. 2018. Intel(R) Software Guard Extensions (Intel(R) SGX) – Developer Guide. (2018). https://download.01.org/intel-sgx/linux-2.3.1/docs/Intel_SGX_Developer_Guide.pdf

[66] Intel Corporation. 2018. Intel(R) Software Guard Extensions (Intel(R) SGX) SDK for Linux* OS – Developer Reference. (2018). https://download.01.org/intel-sgx/linux-2.3.1/docs/Intel_SGX_Developer_Reference_Linux_2.3.1_Open_Source.pdf

[67] Intel Corporation. 2018. L1 Terminal Fault. (2018). https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

[68] Intel Corporation. 2018. PoET 1.0 Specification – Sawtooth v1.1.2 documentation. (2018). https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html

[69] Intel Corporation. 2018. Resources and Response to Side Channel L1 Terminal Fault. (2018). https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html

[70] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE S&P*.

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

[72] K. Järvinen, V. Kolesnikov, A. Sadeghi, and T. Schneider. 2010. Efficient Secure Two-Party Computation with Untrusted Hardware Tokens. In *Towards Hardware-Intrinsic Security - Foundations and Practice*.

[73] K. Järvinen, V. Kolesnikov, A. Sadeghi, and T. Schneider. 2010. Embedded SFE: Offloading Server and Network Using Hardware Tokens. In *FC*.

[74] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*.

[75] J. Katz and L. Malka. 2011. Constant-Round Private Function Evaluation with Linear Complexity. In *ASIACRYPT*.

[76] J. Katz and L. Malka. 2011. Constant-Round Private Function Evaluation with Linear Complexity. In *ASIACRYPT*.

[77] Á. Kiss, M. Naderpour, J. Liu, N. Asokan, and T. Schneider. 2019. SoK: Modular and Efficient Private Decision Tree Evaluation. *PoPETs* 2019, 2 (2019), 187–208.

[78] Á. Kiss and T. Schneider. 2016. Valiant's Universal Circuit is Practical. In *EUROCRYPT*.

[79] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*.

[80] P. Koeberl, V. Phegade, A. Rajan, T. Schneider, S. Schulz, and M. Zhdanova. 2015. Time to Rethink: Trust Brokerage using Trusted Execution Environments. In *TRUST*.

[81] V. Kolesnikov and T. Schneider. 2008. A Practical Universal Circuit Construction and Secure Evaluation of Private Functions. In *FC*.

[82] V. Kolesnikov and T. Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*.

[83] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE S&P*.

[84] K. A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, and R. Ankele. 2016. Exploring the Use of Intel SGX for Secure Many-Party Applications. In *Workshop on System Software for Trusted Execution (SysTEX)*.

[85] J. Lind, I. Eyal, P. R. Pietzuch, and E. G. Sirer. 2016. Teechain: Payment Channels Using Trusted Execution Environments. In *arXiv preprint 1612.07766*.

[86] Y. Lindell. 2018-08-16. The Security of Intel SGX for Key Protection and Data Privacy Applications. (2018-08-16). https://cdn2.hubspot.net/hubfs/1761386/security-of-intelsgx-key-protection-data-privacy-apps.pdf

[87] H. Lipmaa, P. Mohassel, and S. Sadeghian. 2016. Valiant's Universal Circuit: Improvements, Implementation, and Applications. In *Cryptology ePrint Archive, Report 2016/017*.

[88] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.

[89] J. Liu, M. Juuti, Y. Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*.

[90] J. Loftus and N. P. Smart. 2011. Secure Outsourced Computation. In *AFRICACRYPT*.

[91] M. Marlinspike. 2018. Technology Preview: Private Contact Discovery for Signal. (2018). https://signal.org/blog/private-contact-discovery/

[92] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[93] M. Milutinovic, W. He, H. Wu, and M. Kanwal. 2016. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *Workshop on System Software for Trusted Execution (SysTEX)*.

[94] A. Moghimi, T. Eisenbarth, and B. Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *CT-RSA*.

[95] A. Moghimi, G. Irazoqui, and T. Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*.

[96] P. Mohassel and S. Sadeghian. 2013. How to Hide Circuits in MPC An Efficient Framework for Private Function Evaluation. In *EUROCRYPT*.

[97] P. Mohassel and S. S. Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *EUROCRYPT*.

[98] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. V. Lokam, E. Shi, and V. Goyal. 2017. HOP: Hardware makes Obfuscation Practical. In *NDSS*.

[99] S. Niksefat, B. Sadeghiyan, P. Mohassel, and S. Sadeghian. 2014. ZIDS: A Privacy-Preserving Intrusion Detection System Using Secure Two-Party Computation Protocols. In *The Computer Journal*.

[100] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*.

[101] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC*.

[102] D. A. Osvik, A. Shamir, and E. Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*.

[103] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. 2019. XONN: XNOR-based Oblivious Deep Neural Network Inference. In *USENIX Security*.

[104] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIACCS*.

[105] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.

[106] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. 2018. DeepSecure: Scalable Provably-Secure Deep Learning. In *DAC*.
[107] M. Russinovich. 2017. Announcing the Confidential Consortium Blockchain Framework for Enterprise Blockchain Networks. (2017). https://azure.microsoft.com/en-us/blog/announcing-microsoft-s-coco-framework-for-enterprise-blockchain-networks/
[108] M. Russinovich. 2017. Introducing Azure Confidential Computing. (2017). https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
[109] T. Schneider and O. Tkachenko. 2018. Towards Efficient Privacy-Preserving Similar Sequence Queries on Outsourced Genomic Databases. In *WPES*.
[110] T. Schneider and O. Tkachenko. 2019. EPISODE: Efficient Privacy-PreservIng Similar Sequence Queries on Outsourced Genomic DatabasEs. In *ASIACCS*.
[111] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE S&P*.
[112] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
[113] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*.
[114] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*.
[115] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *ASIACCS*.
[116] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. 2017. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*.
[117] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. 2015. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S&P*.
[118] O. Tkachenko, C. Weinert, T. Schneider, and K. Hamacher. 2018. Large-Scale Privacy-Preserving Statistical Computations for Distributed Genome-Wide Association Studies. In *ASIACCS*.
[119] C.-C. Tsai, D. E. Porter, and M. Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*.
[120] L. G. Valiant. 1976. Universal Circuits (Preliminary Report). In *STOC*.
[121] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[122] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. 2017. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*.
[123] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*.
[124] N. Weichbrodt, P.-L. Aublin, and R. Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Middleware*.
[125] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. (2018). https://foreshadowattack.eu/foreshadow-NG.pdf
[126] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*.
[127] A. C. Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*.
[128] Y. Yarom and K. Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[129] S. Zahur, M. Rosulek, and D. Evans. 2015. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT*.
[130] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *CCS*.
[131] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*.
[132] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*.

A INTEL SGX SIDE-CHANNEL ATTACKS AND MITIGATION TECHNIQUES

In this section, we review existing side-channel attacks and mitigation techniques on Intel SGX. The Intel SGX website[9] lists some of the most relevant works on side-channel attacks. Furthermore, Lindell [86] gives a good general overview while Wang et al. [123] 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*, i.e., its execution time depends on secret data, such data can be deduced [24, 25]. Furthermore, if different processes are executed on the same machine, secrets can be leaked since the processes share physical resources, e.g., 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. Researchers have shown that cache attacks can be performed across VMs running on different cores or even different CPUs of the same machine [70, 102, 105, 131]. The different categories of side-channel attacks that have been shown to be practical against Intel SGX are briefly explained in the following.

A.1 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 [126] and bits of encryption keys from cryptographic implementation libraries [115]. Later, it has been shown in [122, 123] 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, a.k.a. *controlled-channel attacks*, exploit secret-dependent control transfers and data accesses. Mitigation strategies include placing sensitive data within the same page [115] or detecting the anomalously high exception or interrupt rate often associated with these attacks as T-SGX [114] and Déjà Vu [32] do. These defenses fall short in the face of more sophisticated attacks that avoid producing too many interrupts [123]. An alternative approach is SGX-Shield [113], 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.

A.2 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 B. Modern Intel CPUs have a three-level cache hierarchy. Each core has its own L1 and L2 caches while the L3 cache, which is

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

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 [36].

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 re-used, include Evict+Time, Prime+Probe [102], and Flush+Reload [128]. Recent works [20, 48, 54, 95, 112] 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 accessing the data that he loaded into the cache. From the measured access times the attacker can conclude 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 [20], AES keys [48, 95], and sensitive information such as genomic data [20], images, and parts of text documents [54] from cryptographic and application libraries running inside SGX 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 [112].

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 [114] and Déjà Vu [32]. These defenses do however also impose a noticeable overhead. T-SGX and Déjà Vu rely on Intel *transactional synchronization extension* (TSX), which provides support for *hardware transactional memory* (HTM). HTM enables multiple threads to optimistically execute transactions in parallel, to abort, and roll 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 [50], 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. However, Cloak requires the developer to annotate sensitive data structures manually. Moreover, Intel TSX is not supported by all Intel SGX-enabled processors.

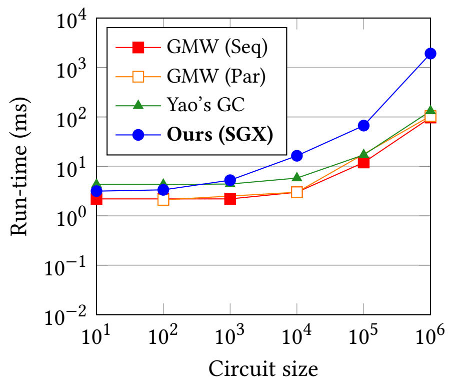
An alternative approach that has been proposed is *data location randomization for SGX*, in short DR.SGX [18]. It prevents information leakage due to secret-dependent data accesses by continuously randomizing the enclave's memory layout and thereby obfuscating the link between memory locations and data objects.

Since cache attacks often target cryptographic libraries, great efforts have been put in designing side-channel resistant, constant-time variants of encryption algorithms [22, 102]. The recently proposed MemJam attack [94] demonstrates the vulnerability of a constant-time AES implementation from the Intel IPP library, which is part of the Intel SGX SDK. The attack has a 4-byte intra-cache line granularity, breaking the assumption that constant cache line accesses prevent leakage. Only code with constant time and constant memory accesses can be expected to be leakage-free. It is suggested in [94] to exclusively use hardware-based or hardware-assisted implementations such as AES-NI. However, unfortunately hardware support for cryptographic primitives is limited and support for AES-NI is in some cases disabled in the BIOS.

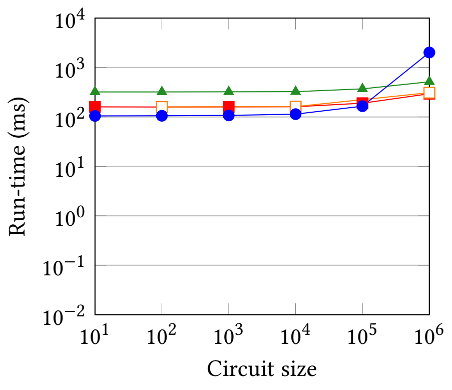
A.3 Speculative Execution-Based Attacks

Speculative execution is an optimization technique used by modern processors. The Meltdown [88] and Spectre [79] 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 whose direction is yet to be determined. This might be either because the direction depends on preceding instructions, or because the instruction is being executed *out-of-order* to further speed up the program. When this happens, the processor makes 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 [79] attacks including SgxPectre [30] trick the processor into speculatively executing instruction sequences that are not part of the victim's intended execution path, and information that the victim (e.g., enclave) is authorized to access is leaked. Similarly, Meltdown [88] and Meltdown-type attacks such as Foreshadow [67, 121, 125] exploit 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 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. One of its three variants 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 by measuring the time it takes to reload each slot of the oracle buffer.

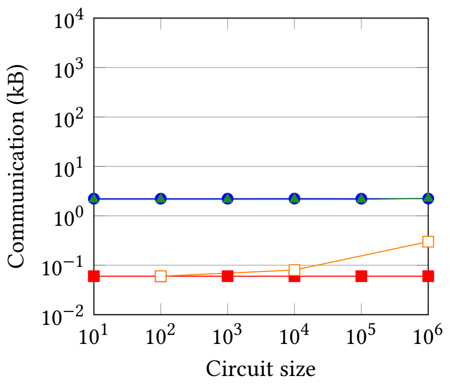
Spectre-type attacks against Intel SGX require vulnerable code to be executed within the enclave. This is not the case for Meltdown-type attacks, which can even be performed without executing the victim enclave. Foreshadow was able to extract enclave secrets residing in protected memory or CPU registers but more importantly, it was the first to extract long-term keys from Intel-provided architectural enclaves such as the quoting enclave, thereby completely invalidating remote attestation guarantees. It showed that Meltdown-type attacks can also be used to breach non-hierarchical intra-address space isolation barriers. The original Meltdown attack was used to breach the memory isolation barriers between kernel and user space, allowing an unprivileged attacker to read kernel memory. It has been mitigated using kernel page table isolation techniques [51], which, however, cannot defend against Foreshadow. Intel has released a microcode update to protect enclaves



(a) Total Run-times in LAN

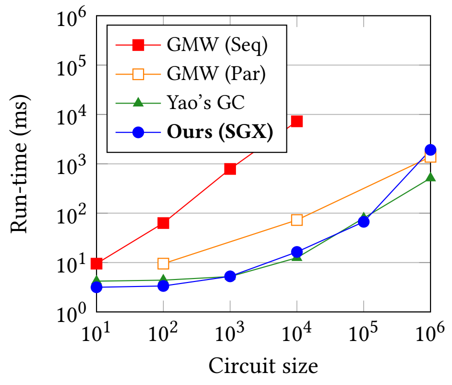


(b) Total Run-times in WAN

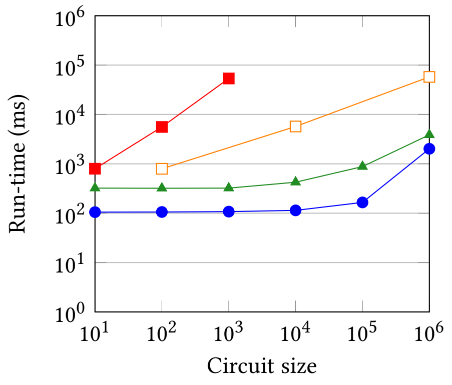


(c) Total Communication

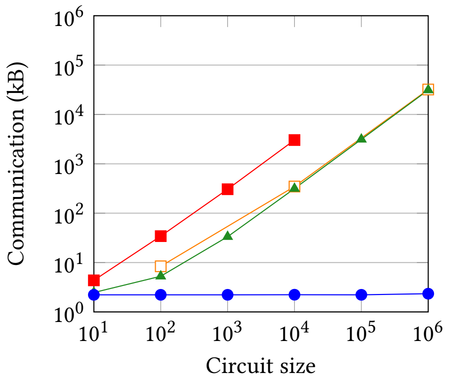
Figure 5: Performance comparison of SFE protocols for X circuits; for the Intel SGX-based and Yao's GC protocol differences between sequential ("seq") and parallel ("par") circuits are negligible and therefore omitted.



(a) Total Run-times in LAN



(b) Total Run-times in WAN



(c) Total Communication

Figure 6: Performance comparison of SFE protocols for A circuits; for the Intel SGX-based and Yao's GC protocol differences between sequential ("seq") and parallel ("par") circuits are negligible and therefore omitted.

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 *hyper-threading* (HT), Intel's proprietary implementation of *simultaneous multithreading* (SMT), is enabled. Intel has acknowledged the possible threat by deriving different keys depending on whether HT is enabled or disabled and includes its status in quotes. Consequently, service providers can decide whether or not to reject attestations from HT-enabled platforms. In the long run, in future Intel processors, Spectre, Meltdown, and Foreshadow will reportedly be mitigated through hardware changes [67, 69, 125].

A.4 Combined Countermeasures

Two recent works, HyperRace [31] and Varys [101], aim to offer protection against a wider range of side-channel attacks. Both defend against interrupt-based as well as HT-based attacks. HT improves processor performance by allowing to run two concurrent threads on a single physical CPU core with two *logical* cores. Since these share all the core resources, HT enables or assists same-core side-channel attacks, e.g., [49, 123]. HT-based attacks typically do not trigger a large number of interrupts and are therefore not detected by many existing defenses. Nonetheless, due to its performance gains, simply disabling HT is not 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 threads. Moreover, both works additionally monitor interrupts and therefore make page table-based side-channel attacks as well as L1/L2 cache attacks on enclaves more difficult or even impossible to mount. However, they do not protect against cross-core side-channel attacks such as LLC attacks. Cross-core side channels however tend to be noisy and are often difficult to exploit in practice.

Unfortunately, assuming that developers can write code that is resistant to all sorts of side-channel attacks is unreasonable. The recent MemJam attack [94] shows that even code written by experts can be vulnerable. Additionally, some attacks like Foreshadow [121, 125] do not require code vulnerabilities in the victim enclave.

Side-channel resistance is especially important for cryptographic implementations to avoid leaking secret keys. Hardware-assisted implementations such as AES-NI have so far withstood attacks, therefore their usage is strongly encouraged. Though there is in-enclave support for AES-NI and the SGX Developer Guide [65] references it as being resistant to timing side-channel attacks, the Linux SGX SDK does not include it [9, 48], instead using a slower software implementation without side-channel mitigations. Linux developers can manually link the SDK with a precompiled optimized binary of the Intel IPP library, which uses AES-NI, improving both the performance and the security of their SGX applications [57, 94].

B EXTENDED RUN-TIME & COMMUNICATION COMPARISON

In Fig. 5 and Fig. 6 we compare the run-times (LAN and WAN) and the communication of our Intel SGX-based SFE protocol to the implementations of Yao's GC and the GMW protocol provided by the ABY framework [41]. Specifically, we compare the performance results for the X and A circuits (cf. §5.1), which contain only XOR and AND gates and therefore constitute the best and worst case scenario for cryptographic SFE protocols, respectively. It should however be noted that these extreme cases usually do not appear in practical applications.

Unsurprisingly, the optimized ABY implementation outperforms our Intel SGX-based prototype in the LAN setting since evaluating XOR gates is essentially "free" for both cryptographic SFE protocols (cf. §2.1). In Fig. 5, the communication for all cryptographic SFE protocols is lower than for our Intel SGX-based protocol, which has optimal communication complexity. This visualizes the somewhat constant overhead caused by TCP and TLS (since ABY does not measure TCP overhead, cf. §5.3) as well as by employing Google protobuf (cf. §4.2).

In the worst case for cryptographic SFE protocols, i.e., evaluating only AND gates, we can report even higher improvements than for the average case (cf. §5.4): in the WAN setting, we improve run-time by factor 1.9x over Yao's GC and by factor 19x over the GMW protocol for a circuit with 1 million AND gates; the communication is reduced from 30.52 MB / 31.26 MB to 2.47 kB, respectively.

For the Intel SGX-based SFE protocol, the measured difference in run-times between the A, X, and AX circuits is statistically insignificant. This is expected since evaluating AND and XOR gates differs only in performing the respective AND and XOR operation on machine code level. As a consequence, when using tools to compile high-level functions to Boolean circuits, the total number of gates in the circuit should be minimized. In contrast, currently existing SFE compilers that produce output for Yao's GC and the GMW protocol try to minimize the total number of AND gates and in case of the interactive GMW protocol also the AND depth [26, 27, 60].