Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation

Debayan Gupta[1], Benjamin Mood[2], Joan Feigenbaum[1],
Kevin Butler[2], and Patrick Traynor[2]

[1] Yale University. E-mail: {debayan.gupta, joan.feigenbaum}@yale.edu
[2] University of Florida. E-mail: bmood@ufl.edu, {butler,traynor}@cise.ufl.edu

Abstract. Recent developments have made two-party secure function evaluation (2P-SFE) vastly more efficient. However, due to extensive use of cryptographic operations, these protocols remain too slow for practical use by most applications. The introduction of Intel's Software Guard Extensions (SGX), which provide an environment for the isolated execution of code and handling of data, offers an opportunity to overcome such performance concerns. In this paper, we explore the challenges of achieving security guarantees similar to those found in traditional 2P-SFE systems. After demonstrating a number of critical concerns, we develop two protocols for secure computation in the semi-honest model on this platform: one in which both parties are SGX-enabled and a second in which only one party has direct access to this hardware. We then show how these protocols can be made secure in the malicious model. We conclude that implementing 2P-SFE on SGX-enabled devices can render it more practical for a wide range of applications.

1 Introduction

Secure Function Evaluation (SFE) is a powerful way to protect sensitive data. Made possible by a range of cryptographic primitives, SFE allows multiple parties to compute the result of a function without revealing the potentially sensitive inputs of any individual party. In this paper, we focus on the case of two-party secure function evaluation (2P-SFE). While significant advances in both the security provided by and the performance of these underlying primitives has improved dramatically over the past decade [1,9,17,25,27,29,33], the expense of using 2P-SFE remains too high for most practical applications.

An emerging hardware primitive may help to dramatically reduce the cost of such computation. Intel's Software Guard Extensions (SGX) [2,26] provide a module within upcoming chipsets that allow for the creation of secure containers called "enclaves". These hardware-enforced sandboxes allow for code and data to be executed without the influence of code running in the traditional registers of the processor. In addition, an SGX system can use hardware-based attestation to prove that an enclave performs the operations as claimed. While not necessarily appropriate for all scenarios, this set of capabilities may help to support the use of fast and strong 2P-SFE in a wide range of practical applications.

In this paper, we perform the first analysis of SGX as a platform on which to implement 2P-SFE. Beginning with a tutorial example, we show why the naive execution of functions within SGX fails to provide the strong properties necessary to prevent significant leakage. From this observation, we then make the following contributions:

- We show how to augment an SGX system to provide stronger guarantees against leakage and provide a protocol that enables two SGX systems to perform 2P-SFE efficiently when compared to a purely garbled circuits implementation. We refer to this approach as *SGX-supported 2P-SFE*. We then provide a protocol for securely outsourcing the SGX-supported 2P-SFE computation from a resource constrained device (*i.e.*, one without an SGX module) to an SGX compliant device (*i.e.*, another device that has an SGX module). This allows us to take advantage of a remote SGX hardware unit without requiring universal deployment.
- We show how to modify 2P-SFE protocols secure against semi-honest adversaries so that, when run on augmented SGX machines, they are secure against malicious adversaries.
- We describe a number of novel use cases for SGX with our augmentations.

The rest of the paper is organized as follows: Section 2 provides background on 2P-SFE and SGX. Section 3 explains problems that arise in straightforward attempts to use SGX for 2P-SFE. Section 4 describes how to augment SGX so that it can be used to implement 2P-SFE, a secure outsourcing protocol for non-SGX machines, and how 2P-SFE and SGX can be efficiently used in conjunction to provide better security. Section 5 discusses previous work on secure-execution environments, and Section 6 provides conclusions and open questions.

2 Technical Background

We begin with a brief overview of garbled-circuit 2P-SFE and SGX. We use this as a point of departure for our investigation of SGX-based protocols for 2P-SFE and why they are harder to design than one might imagine at first glance.

2.1 Garbled Circuits for Two-Party, Secure Function Evaluation

In a garbled-circuit protocol, two parties with private inputs jointly compute a function represented as a Boolean circuit. Both parties receive outputs – the scenario described in Section 1, which has a single output y for both parties, is a special case; in general, the protocol may deliver different outputs to each party. First, a compiler [33,38] is used to convert the function into a Boolean circuit. One of the parties, the *generator*, encrypts, or *garbles* the Boolean circuit. He then sends it to the *evaluator*, who evaluates the garbled circuit without learning any information about the generator's inputs, intermediate values (*i.e.*, those computed by non-output gates of the circuit), or the generator's output. Finally, the evaluator sends the generator's (encrypted) output back to him.

Each gate in a Boolean circuit can be evaluated using its truth table to get the output corresponding to the input values. Likewise, a garbled circuit is made up of many garbled gates, and each gate is evaluated in turn. A garbled gate's output entry in the truth table is encrypted under a unique combination of the two inputs: $TT_{i,j} = Enc(X_i, Y_j) \oplus Out_{i,j}$, where $TT_{i,j}$ is the truth-table entry created by the i^{th} value of wire X and the j^{th} value of wire Y, and $Out_{i,j}$ is the corresponding unencrypted output value. The truth-table entries are permuted so that the position of the (only) decryptable entry does not leak the underlying Boolean value. Once the evaluator receives the garbled gates and the input values, she finds the correct garbled output by trying to decrypt each truth-table entry or by using the point and permute optimization [33].

There are two basic types of adversaries in the garbled-circuit literature: semi-honest and malicious adversaries; each captures a basic threat model. (There exist others, such as the covert model, but we do not discuss them here.) Semi-honest adversaries faithfully follow the protocol, but attempt to gain information by observing all transmitted messages. Malicious adversaries, on the other hand, may behave in any arbitrarily manner in an attempt to gain information about another party's input or output, to corrupt the computation (*i.e.*, to cause incorrect outputs), or to block the protocol execution from completing.

To achieve security against malicious adversaries, the computation must be performed N times in order to prevent the generator from creating an incorrect circuit. The security parameter N sets the upper bound on an adversary's successfully cheating at $\frac{1}{2^N}$. There must be mechanisms to ensure that the same inputs are used each time and a way to ensure the evaluator does not corrupt the generator's output. These are solved problems in the garbled-circuit literature.

2P-SFE and garbled circuits were introduced in the seminal paper of Yao [50], and the area has since been studied extensively by the cryptography community. One very notable achievement was the creation of the first general-purpose 2P-SFE platform, Fairplay [33]. Today, many 2P-SFE platforms exist [1,9,17,25,27,29,32,37], and their performance is improving. Such platforms have been used for scenarios as varied as those of farmers conducting beet-root auctions [8], inter-domain routing [24], governments reporting aggregated salary data [7], and database policy compliance [15]. For a detailed explanation of many essential garbled-circuit techniques, see Kreuter *et al.* [29] and Perry *et al.* [41].

2.2 Intel's Software Guard Extensions Module

The Software Guard extensions (SGX) module allows parts of programs to be executed inside of separate segments of the CPU called *enclaves*. This is a general-purpose module (unlike, say, a DRM module). SGX provides a hardware-based guarantee that the programs and memory inside an enclave cannot be read or modified from outside of the enclave (including a program in different enclave). In particular, neither `root` nor any other type of special-access program can read or modify the memory inside an enclave. Technically, the data inside of an enclave are still within the same registers and cache as other programs; however, SGX processors provide functionality to prevent unauthorized access.

An adversary should not be able to determine what is accessed inside of the enclave or what is written back to RAM when the cache is full. Therefore, any data in the enclave that must be written back to main memory is encrypted and signed so that it cannot be read or modified by another program. Modifications to code, data, or stack outside an enclave cannot interfere with the operation of the enclave except in one way: if something needed by a program in the enclave is simply unavailable or has been corrupted, then the program may have to abort.

Comprehensive overviews of SGX can be found in Intel's whitepapers [2,26]. Design of systems and protocols that make extensive use of SGX is covered by, *e.g.*, Baumann *et al.* [6] and Schuster *et al.* [43].

2.3 Towards Using Secure Hardware for Garbled-Circuit Protocols

Both garbled circuits and SGX are designed for scenarios in which parties have private input data for a computation where they want to receive the result of the computation while no one else learns either the input or the result. Therefore, it is natural to consider using SGX-enabled machines to execute a garbled-circuit protocol. The reason that it is not straightforward to do so is that garbled circuits and SGX use different techniques to protect private inputs.

In garbled-circuit protocols (and SFE more generally), cryptographic guarantees are used to ensure the privacy of the data. In SGX, users rely on secure hardware to guarantee data privacy. SGX provides security against malicious adversaries as long as one trusts Intel's setup process. In the SFE world, this is comparable to having a trusted setup, on top of which one runs one's protocol (here, part of the "setup" occurs at the Intel factory when the hardware and private key are created). The security properties of the exact model used by SGX are described in Intel's whitepapers [2,26].

3 Why Simple "Solutions" Do Not Quite Work

The security guarantees provided by SGX do not immediately translate into being able to perform 2P-SFE protocols in general or even garbled-circuit protocols in particular. Simple solutions that use unmodified SGX primitives may leak information or, in some cases, undermine the security of other code running under SGX. In this section, we explain how that can happen.

3.1 A simple 2P-SFE protocol implemented with SGX

Below, we describe a naive, straw-man protocol for performing SGX-supported 2P-SFE. There exist numerous ways of doing this, but almost all of them suffer from a number of problems that we discuss in the next subsection.

Setup: We start with the standard 2P-SFE setup – two mutually distrustful parties with private inputs who wish to jointly compute a function and produce private results. In this scenario, both parties have SGX-enabled machines and

have agreed to run a specific program. The two parties are as follows: the *evaluator*, who will use his SGX module to evaluate the program, and the *sender*, who will check the agreed upon program and then send her input. In the following, a superscripted "+" denotes a public key, while a superscripted "−" denotes a private key that does not leave the SGX enclave.

Protocol

1. The sender ensures the evaluator will evaluate the correct program, $prog_{sgx}$, by checking the signed measurement, $Ecv^{eval}_{measure}$, from the evaluator's enclave. $Ecv^{eval}_{measure}$ is signed by the evaluator enclave's private key $Ecv^{-}_{key_{eval}}$.
2. The sender encrypts her input, $input_{sender}$, under the evaluator enclave's public key, $Ecv^{+}_{key_{eval}}$, and sends it to the evaluator.
3. The sender's encrypted input, $Enc(input_{sender})$ is decrypted inside of the evaluator's enclave using $Ecv^{-}_{key_{eval}}$.
4. The evaluator enters his own input, $input_{eval}$ into the enclave.
5. The enclave puts $input_{sender}$ and $input_{eval}$ into the SGX program, $prog_{sgx}$. It then executes $prog_{sgx}$ and encrypts the sender's output, $output_{sender}$, under the sender enclave's public key, $Ecv^{+}_{key_{send}}$.
6. The evaluator's enclave releases the evaluator's output to him, and sends the sender's encrypted output, $Enc(output_{sender})$, to the sender.
7. The sender decrypts $Enc(output_{sender})$ using $Ecv^{-}_{key_{send}}$.

3.2 Problems with simple SGX-supported 2P-SFE

Side channels

1. **Runtime:** 2P-SFE protocols are not directly vulnerable to timing attacks. This is achieved by ensuring all program paths take equal time, at the cost of efficiency. In SGX-supported 2P-SFE, if a secret value x determines the number of times, for instance, a loop is executed, the timing could easily narrow the range of x. Principally, an attacker could execute the same program offline with many different iterations of the same loop inside of the enclave to see how long several different numbers of iterations take. This may provide a lot of information if each iteration of the loop is easily identifiable, *e.g.*, if each iteration takes a second to execute.
2. **RAM Access:** Data access is not hidden in SGX-supported 2P-SFE, which can potentially leak significant amounts of data. *E.g.*, a simple database style query using a binary search, where one side, the client, sends a private query to check whether a given value exists within the database. The enclave on the server reads in the plaintext records and matches them, one by one, to the queried value. In such a scenario, the data access alone is enough leak information about the queried value. (If the query matched, we have the value itself, if not, we know that it lies within a certain range.) There exist some methods to add hardware-level cryptographic support to FPGAs [45], but not for RAM. The best ways to make RAM secure are still Oblivious RAM and similar techniques [20].

3. **RAM Timing:** A timing attack could reveal a lot of information about the item being queried in the binary search. If the item is located on the first jump, we know that it's the value in the middle, etc.

Cryptography vs Memory Out of Bounds Garbled circuits rely upon cryptography for data privacy – information leakage is not an issue as we have proofs for correctness and security. While it is theoretically possible to "leak" data by simply outputting it in the predefined program, such a blatant problem is easy to notice. SGX, if used improperly, might leak information if memory goes out of bounds, which is one of the most common bugs in everyday programming [43]. This is among the most frequent errors in programming, and can have catastrophic consequences [48,14]. Unfortunately, in SGX, such an error would not only break the security of the program (and enclave) in question, but would also affect the security of SGX as a whole, since users might be able to access or modify data that they should not be able to see.

Trusting SGX vs Trusting Cryptography SGX requires the users to trust that the evaluator of the program has not broken into the enclave to watch the memory and that the supply chain was not disrupted with insecure parts. These might not be acceptable assumptions for nation states or large companies. In contrast, 2P-SFE protocols provide cryptographic guarantees. They prove themselves equivalent to the "ideal model," which uses a trusted third party. SGX uses the trusted platform model, which is weaker than the trusted third party model and allows side-channel and information flow attacks.

SGX requires us to have trust in hardware and standard cryptographic primitives (which are used by SGX to protect data), while a 2P-SFE protocol needs only the latter. Moving the "trust" from software to hardware presents additional problems – the authors are unaware of any techniques that could be used to sign and verify hardware. Given the recent issues with nation states actively infiltrating hardware vendors at massive scales for bulk data collection, this is a major problem. Ultimately, the trust in SGX boils down to trust in hardware suppliers and whether or not the hardware can be opened and the CPU read.

4 Using SGX for 2P-SFE Computations

Having outlined the capabilities and limitations of SGX-supported 2P-SFE, we now present our solutions to the problems faced when trying to use SGX for 2P-SFE protocols.

4.1 Using SGX for 2P-SFE: Problems and solutions

Our solution is to augment the SGX programs to prevent (or reduce) data leakage in SGX for 2P-SFE computations. These augmentations are described below.

Timing Side Channel: We must ensure all code-paths take approximately the same amount of time. There are many such obfuscation-based palliative mechanisms, as well as general mitigation strategies [34]. However, these problems are

more complex in some scenarios – *e.g.*, when a secret variable determines how many times a loop executes. In this case, the time the program takes can reveal information about the value of the secret variable. It is possible to prevent any secret values from being revealed by having a fixed loop bound, however, this may not always be preferable. We can limit the amount of information leaked when executing a loop by including a pseudo-random N extra loop iterations - where N is based upon secret information from both parties. Using this technique, neither party learns the number of iterations executed.

Memory Side Channel: We must ensure all memory that can be touched by the SGX program is touched one single time at the beginning of the program. Once the SGX program touches a piece of plaintext memory, the memory should not be read again unless the read is not dependent on secret information. If the read is dependent on secret information, the evaluator may be able to learn something about the secret [18,39]. However, if we need too much data and some are encrypted and stored outside of the enclave, there might be a correlation between when a block of memory is read and when a block of encrypted memory is sent back to RAM; *e.g.*, if a binary search program that runs inside an enclave reads one element at a time, mere observation yields the secret query (within a range, if it is missing). In order to prevent this problem, we must ensure a *mix* operation is performed that removes any correlation between plaintext memory and encrypted memory; *e.g.*, this would occur if the memory was placed outside of the enclave in the same order as it was entered. Such *mix* operations, which continuously shuffle and re-encrypt data as they are accessed, already exist, and are widely used to implement Oblivious RAM [20,46].

Array Out of Bounds: To mitigate the risk of arrays out of bounds in SGX, we apply safe memory access techniques to ensure memory does not go out of bounds. SGX programs can use bound-checking data structures or memory safe languages [43]. Although such techniques slow down the execution time of the application, both of the aforementioned methods would still be significantly faster than executing the programs in a 2P-SFE protocol.

Cost of a 2P-SFE protocol vs SGX: In Table 1, we note the expected cost of normal 2P-SFE using garbled circuits and SGX-supported 2P-SFE. We examine the costs of setup, input, the operation itself, data access, and memory access. As shown in the table, the primary reason for the expected increase in the speed of SGX-supported 2P-SFE over a garbled-circuit protocol is the amount of cryptography required for each operation and data access in 2P-SFE (which is free in SGX). However, unlike garbled-circuit protocols, SGX encounters a cost to push memory out of the cache to RAM (*Non-Cache Access*).

4.2 Half and Half

With the techniques above, 2P-SFE protocols and SGX can be used together in scenarios where parties trust each other enough to want to cooperate in the first place but not enough to release private data or blindly trust the other parties not to cheat [30]. However, when different groups of parties want to perform a secure computation together, a user may trust one group over another; the different

	2P-SFE$_{Semi}$		2P-SFE$_{Malicious}$		SGX	
	Sym	Asym	Sym	Asym	Sym	Asym
Setup	-	-	-	-	$O(1)$	$O(1)$
Input	$O(N)$	$O(K)$	$O(N*S)$	$O(K*S)$	$O(N)$ $^{+}$	-
Per Operation[1]	$O(1)$	-	$O(S)$	-	-	-
Data (array) Access	$O(N)$	-	$O(N*S)$	-	-	-
Non-Cache Access[2]	-	-	-	-	$O(1)$	-

Table 1: Cost (in terms of cryptography) for operations in 2P-SFE and SGX-supported 2P-SFE. " - " means there is no cryptography required. N is length of input. C is length of the circuit/program. K is the bit-security parameter. S is the stat parameter (number of circuits in 2P-SFE). [1] - per gate for 2P-SFE and per processor instruction for SGX-supported 2P-SFE. [2] - the cost of saving and loading a value to or from main memory for SGX. $^{+}$ - assumes we attained a symmetric key during the setup phase and used it to encrypt the input.

guarantees and characteristics of SGX-supported 2P-SFE and current 2P-SFE protocols mean that it might make sense to use one technique for a certain group but not another. We now examine how to perform a secure computation where one part of it is evaluated using current 2P-SFE protocols and the other is evaluated using SGX-supported 2P-SFE.

We start with two companies, A and B (as shown in figure 1), which want to perform a secure computation involving nodes both inside and outside their private networks. Parts of the computation are done inside of each company, while others require A and B to cooperate. Thus, companies could use the trust model of SGX when within their own networks and 2P-SFE when they want cryptographic guarantees instead of assuming that the hardware remains secure.

To perform such hierarchical or "mixed" SGX computations, users need to know how to convert a value from a 2P-SFE protocol to an SGX-supported 2P-SFE value and vice-versa. Once we know how to perform these transformations, we can run "mixtures" of 2P-SFE protocols and SGX. For simplicity, we deal with the semi-honest setting, although we note there are ways to do the same conversions in the malicious setting. For the purposes of this short protocol, the evaluator is the evaluator in both 2P-SFE and SGX-supported 2P-SFE. The generator is the generator for 2P-SFE and the sender in SGX-supported 2P-SFE.

Before we briefly describe the conversion process, we describe more about garbled circuits. During the evaluation of the garbled circuit each wire holds an encrypted value. The generator knows the possible encrypted values (that is, which values represent 0 and 1), but does not know which value is actually on the wire (the value the evaluator has). The evaluator knows the encrypted value on each wire value, but does not know what any value represents. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity.

Conversion from Garbled Circuit to SGX:

1. For each garbled wire w_i we will convert to an SGX value, the evaluator has w_i^r (the encrypted result) and the generator has w_i^0 and w_i^1 (the encrypted values that represent 0 and 1).

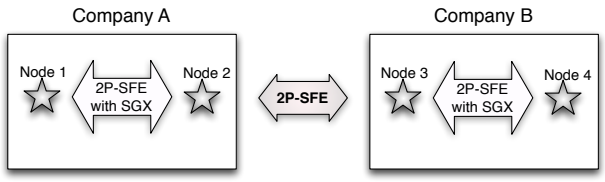

Fig. 1: Half and Half. In this usage, we convert SGX-supported 2P-SFE values to standard 2P-SFE values and back in order to take advantage of the speed of the combined form when the trust model is acceptable and still allow for a stronger model when the trust model of SGX-supported 2P-SFE is not acceptable (say, the user does not trust Intel when using a public network).

2. The generator enters w_i^0 and w_i^1 into $prog_{sgx}$ (the SGX program) as input.
3. The evaluator enters in w_i^r into $prog_{sgx}$ as her input.
4. $prog_{sgx}$ calculates whether w_i^r is w_i^0 or w_i^1 and sets the corresponding input, b_i, to match w_i^r.
5. $prog_{sgx}$ uses each b_i as input.

Conversion from SGX to Garbled Circuit:

1. For each bit b_i that will be converted into a garbled value w_i, the generator creates both possible garbled values, w_i^0 and w_i^1, that will represent the two possible values of b and enters them into $prog_{sgx}$.
2. $prog_{sgx}$, based on whether b_i is a 0 or 1, selects either w_i^0 or w_i^1 to be w_i^r.
3. Each w_i^r is sent to the evaluator to be used as input to the garbled circuit.
4. The generator uses his values, w_i^0 and w_i^1, in the creation of the garbled circuit to ensure w_i^r will map to a value.

Security: In order for either the generator or evaluator to learn additional information, they have to (1) posses either w_i^0 or w_i^1 and posses w_i^r, or (2) see b_i outside of the enclave. Since b_i only exists inside of the enclave, it will not be seen by either the generator or evaluator. The generator only ever sees w_i^0 and w_i^1 and never sees w_i^r. Likewise, the evaluator only sees w_i^r and never sees w_i^0 or w_i^1. Thus, neither party will learn any additional information.

4.3 Outsourcing

For devices that do not have an SGX module (or are slow), it would be useful to have the ability to securely outsource computation to a more powerful or better equipped system. There have already been a number of works addressing this situation in 2P-SFE [10,11,12,13,36]. In this section, we examine how we can outsource from a constrained device (that does not possess an SGX module) when we want to perform SGX-supported 2P-SFE.

In our setup, seen in figure 2, the sender does not have an SGX unit and is outsourcing to a server, the cloud, who has an SGX unit. Any outsourcing protocol must guarantee (1) the party we are outsourcing to (the cloud) cannot

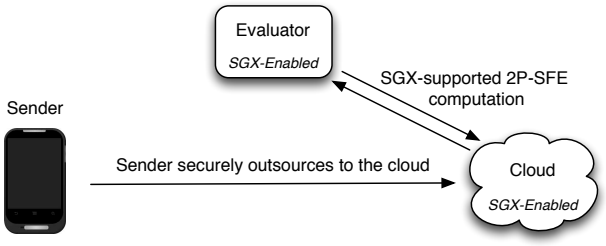

Fig. 2: Outsourcing. Shows the different parties in our outsourcing protocol.

cheat, and (2) the party that performs the SGX execution (the other party in the original SGX-supported 2P-SFE computation, the evaluator) cannot cheat.

We assume that we are trying to protect the input and output of the sender; we also assume the cloud and evaluator do not collude, *i.e.*, they are not working together to corrupt the sender's output or input. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity. As before, superscripted "+" and "−" signs denote public and private keys, respectively.

Protocol:

1. The cloud and evaluator perform the standard SGX setup to initialize their SGX units and confirm they are running the desired program.
2. Both parties pass enclave public keys, $Ecv^{+}_{key_{cloud}}$ and $Ecv^{+}_{key_{eval}}$ to the sender and authenticate by using MRSIGNER [2,26].
3. Both the evaluator and cloud enclaves send to the sender their enclave measurements, $Ecv^{cloud}_{measure}$ and $Ecv^{eval}_{measure}$.
4. The sender checks the measurements $Ecv^{cloud}_{measure}$ and $Ecv^{eval}_{measure}$ are correct.
5. The sender encrypts his input, $input_{sender}$, and a public key for his output, Out^{+}_{key}, under $Ecv^{+}_{key_{eval}}$ to create $Enc(input_{sender}||Out^{+}_{key})$ and sends it to the cloud.
6. The cloud enters $Enc(input_{sender}||Out^{+}_{key})$ into the SGX program, $prog_{sgx}$. *We note here there is no reason the cloud cannot also have input into the program, if it is desired.*
7. The input is sent from the cloud to the evaluator.
8. $prog_{sgx}$ is run according to the previous SGX-supported 2P-SFE protocol.
9. The sender's output, $output_{sender}$ is encrypted under Out^{+}_{key} as a final step in $prog_{sgx}$.
10. This value, $Enc(output_{sender})$, is sent from the evaluator to the sender.
11. The sender uses the output private key Out^{-}_{key} to decrypt $Enc(output_{sender})$.

Security of the Sender's Data

Input: Since the sender's input is encrypted under the evaluator's enclave private key, it can only be decrypted inside of the evaluator's enclave. Given the measurement of the evaluator's enclave, we also know the program inside of the enclave is correct so it will not pass the input outside the enclave.

Output: Since the sender's output is encrypted inside the enclave during evaluation and is only sent outside when it is encrypted under the sender's public key, only the sender can decrypt and read this output.

4.4 Improving the security of 2P-SFE protocols using SGX

Semi-honest or honest-but-curious protocols guarantee security as long as all parties faithfully follow the protocol. Such protocols are much cheaper in terms of computation cost than those that protect against malicious adversaries, who attempt to gain additional information by any means necessary. We can use SGX for parts of the semi-honest 2P-SFE protocol to gain additional security guarantees without incurring significant overhead. We provide a short, intuitive security sketch; a complete, formal proof is omitted for brevity.

First, we replace the OT in the 2P-SFE protocol with an SGX component that acts like an OT. The SGX OT is a stripped down version of the previously described SGX-supported 2P-SFE protocol. In this program, the 2P-SFE evaluator chooses the encrypted form of the input as in the 2P-SFE protocol. This immediately gives us greater security than the standard semi-honest OT since we are not relying on the parties to behave correctly during the OT (*i.e.*, the SGX unit checks whether the parties are running the correct "OT" program). Note that this does not guarantee fair-release of the result, since a malicious party can still cause us to abort at any point.

Similarly, we can replace the circuit generation and evaluation with an SGX component as well. This SGX-evaluation is the program evaluation component described earlier. While we could use the 2P-SFE OT before this part of the protocol, using the SGX OT component gives us better security. After the input and circuit evaluation components are replaced, we can also replace the output component with the SGX output protocol. Replacing all of these elements leaves us with a protocol that is significantly more secure than the original semi-honest 2P-SFE protocol (since the SGX protocol has checks for when a user is malicious), while remaining much cheaper than a malicious 2P-SFE protocol.

4.5 Universal Programs (Circuits)

A *universal circuit* (UC) is a program that takes another program as input (denoted as UC_{prog}) and then executes it. In a UC for two parties, one party enters UC_{prog} as input while the other party enters the input for UC_{prog}.

However, in 2P-SFE, a UC requires a massive number of array accesses due to the nature of oblivious data access. For each operation in UC_{prog} (*e.g.*, $data[a] = data[b] + data[c]$), the inputs to the operation (*i.e.*, $data[b]$ and $data[c]$) have to be found from all the possible values that could be entered into the instructions – *i.e.* this requires a set of if statements to check whether index value v equals b – unless constraints can be added to UC_{prog}. However, in SGX-supported 2P-SFE, this would be more efficient since array access takes $O(1)$. Thus, UC programs can be efficiently and privately executed in an enclave.

4.6 Novel Use Cases for SGX

Secure data storage: With the advent of cloud and multi-user systems, unauthorized data access is a greater problem than ever before. Our idea is to use

SGX as a gatekeeper: If all reads and writes went through the SGX hardware, we could automatically encrypt and decrypt it based on a user-entered key without the need for a specialized drive. A keyboard could enter the enclave password while skipping the operating system and any keyloggers within. Unlike systems such as BitLocker [19], the key here would remain safe even if the operating system was compromised. For cloud storage, the SGX program would encrypt data before they are sent to the cloud server; it could be implemented so as to be transparent to the end user and obviate the need to trust cloud companies.

User Authentication: SGX offers many new avenues for user authentication. It includes MRSIGNER, which signs the enclave before it is deployed. Group authentication is also possible, using EPID (Enhanced Privacy ID) [2], an extension to the Direct Anonymous attestation scheme used in [22,23]. This allows an enclave to sign communications while maintaining privacy within a group. There is also a "pseudonymous" mode, which relaxes the security slightly, allowing the verifier to know whether it has checked an enclave in the past while still maintaining intra-group anonymity.

Cyber-physical applications: Given the security concerns involved in control systems for sensitive infrastructure (*e.g.*, a nuclear power plant or a hydroelectric dam), improving security is highly desirable. In order to prevent attacks on such systems, the controls could be made accessible only through an enclave that would require all orders to the system to be signed; the current state of the system would also be hidden. Periodic signed updates from the enclave to a "master" control system would prevent the system from being taken offline without the knowledge of the master control system. These strategies would mitigate the threat of hackers breaking into the system and altering code or stealing passwords – this information would exist only inside of the enclaves.

Online Games: Online games are played between multiple users on different machines. In order to reduce bandwidth, many games only transfer events, *e.g.*, the information for each user command. Each machine can then process this independently, but at the cost of each machine knowing the entirety of the game's data, including sensitive information about other players' positions. SGX could be used to protect private data from other gamers. By keeping each gamer's private data inside an enclave, a hacker (or any user who uses a tool to read information normally not available to them) would be unable to gain any private information. The enclave would release such private information to the local machine based on triggers in the code, *e.g.*, when an enemy unit is nearby. Further, we can periodically verify the state of each enclave to prevent cheating.

5 Previous Work on Secure-Execution Environments

In this section, we briefly discuss previous work on the use of specialized software and hardware platforms to enable secure execution of code. However, none of these works provide the same guarantees or address the same scenarios as a 2P-SFE protocol. Various levels of code and data protection have been achieved

using approaches as varied as managed runtime environments (such as Java and .NET), tamper resistant software [4], and microkernels.

Haven [6] is an SGX-based system for executing Windows applications in the cloud. VC3 [43], also based on SGX, allows verifiable and confidential execution of MapReduce jobs in untrusted cloud environments.

Systems such as TrustedDB [5] and Cipherbase [3] use different kinds of trusted hardware to process database queries over encrypted data. There exist several other systems [31,40,47] that use trusted system software (usually a trusted hypervisor) along with specialized hardware to achieve various security and privacy requirements. Some, such as Virtual Ghost [16] and Flicker [35], avoid hypervisors by using specialized kernel-level hardware-isolation mechanisms and time-partitioning between trusted and untrusted operations, respectively. Super-distribution systems for transmission of protected digital data also exist [28]. They decrypt protected data using a key from an authorized clearinghouse and then re-encrypt the data with a locally generated key on the end-user system, ensuring that no one else can use the data. Secure co-processors [44] allow programs to execute securely as long as users can verify that they are dealing with untampered programs and hardware.

Intel has a number of whitepapers on SGX [2,26], as well as previous attempts in the same vein, such as the Trusted Execution Technology [21]. ARM trustzone for Cortex-A processors also provides some similar guarantees and has been used to build embedded linux platforms [49], language runtimes for mobile applications [42], and many other systems.

6 Conclusion

This paper presents the first systematic consideration of Intel's Software Guard Extensions as a platform on which to implement two-party secure function evaluation. We show that careful use of SGX primitives can facilitate extremely efficient 2P-SFE protocols, provide an outsourcing mechanism for machines without an SGX module, and discuss augmentations to SGX which provide stronger guarantees against leakage. We also use SGX to convert 2P-SFE protocols secure against semi-honest adversaries into ones secure against malicious adversaries, and discuss a number of use cases for SGX. As SGX-enabled processors eventually make their way onto the market, future work will include implementations and improvements to the efficiency and security properties of these protocols.

Acknowledgements: The first author was supported in part by DARPA contract FA8750-13-2-0058. The second and fourth authors were supported in part by NSF grants CNS-1540217 and CNS-1540218. The third author was supported in part by NSF grants CNS-1407454 and CNS-1409599. The fifth author was supported in part by NSF grants CNS-1464087 and CNS-1464088. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, or the U.S. Government.

References

1. a. shelat and C. H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of the Annual International Conference on Theory and Applications of Cryptographic Techniques*. Springer, 2011.
2. I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.
3. A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.
4. D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333. Springer, 1996.
5. S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):752–765, 2014.
6. A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
7. D. Bogdanov, R. Talviste, and J. Willemson. Deploying secure multi-party computation for financial data analysis. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2012.
8. P. Bogetoft, D. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Kroigård, J. Nielsen, and *et al.* Secure multiparty computation goes live. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 2009.
9. M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proceedings of the USENIX Security Symposium*, 2010.
10. H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, 7(7):1165–1176, 2014.
11. H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.
12. H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium (SECURITY'13)*, 2013.
13. H. Carter, B. Mood, P. Traynor, and K. Butler. Outsourcing secure two-party computation as a black box (short paper). In *International Conference on Cryptology and Network Security (CANS)*, 2015.
14. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129. IEEE, 2000.
15. G. D. Crescenzo, J. Feigenbaum, D. Gupta, E. Panagos, J. Perry, and R. N. Wright. Practical and privacy-preserving policy compliance for outsourced data. In *Proceedings of the International Conference on Financial Cryptography and Data Security - Workshop on Applied Homomorphic Cryptography*. Springer, 2014.
16. J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.

17. I. Damgård, M. Geisler, M. Kroigård, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography*. Springer, 2009.
18. Ú. Erlingsson and M. Abadi. Operating system protection against side-channel attacks that exploit memory latency. Technical Report MSR-TR-2007-117, Microsoft Research, 2007.
19. N. Ferguson. Aes-cbc+ elephant diffuser: A disk encryption algorithm for windows vista. Technical report, Microsoft, 2006.
20. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
21. J. Greene. Intel trusted execution technology. *Intel Technology White Paper*, 2012.
22. T. C. Group. Trusted platform module main specification (tpm1.0). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2011. [Online].
23. T. C. Group. Trusted platform module library specification (tpm2.0). `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`, 2013. [Online].
24. D. Gupta, A. Segal, A. Panda, G. Segev, M. Schapira, J. Feigenbaum, J. Rexford, and S. Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 37–42. ACM, 2012.
25. W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2010.
26. M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11. ACM, 2013.
27. A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ansi c. In *Proceedings of the Conference on Computer and Communications Security*. ACM, 2012.
28. M. Kawahara. Superdistribution: the concept and the architecture. *IEICE TRANSACTIONS (1976-1990)*, 73(7):1133–1146, 1990.
29. B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the USENIX Security Symposium*, 2013.
30. M. Libicki, O. Tkacheva, C. Feng, and B. Hemenway. *Ramifications of DARPA's PROCEED Program*. RAND, Santa Monica, 2014.
31. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.
32. Y. Lindell and B. Riva. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.
33. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–A Secure Two-Party Computation System. In *Proceedings of the USENIX Security Symposium (SECURITY'04)*, 2004.
34. R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

35. J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.
36. B. Mood, D. Gupta, K. Butler, and J. Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.
37. B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, 2016.
38. B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC'12)*, 2012.
39. D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2006.
40. E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 13–24. ACM, 2013.
41. J. Perry, D. Gupta, J. Feigenbaum, and R. Wright. Systematizing secure computation for research and decision support. In M. Abdalla and R. De Prisco, editors, *Security and Cryptography for Networks*, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer International Publishing, 2014.
42. N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 67–80. ACM, 2014.
43. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc 3: Trustworthy data analytics in the cloud.
44. S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, 1999.
45. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient implementation of rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 334–350. Springer, 2003.
46. E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.
47. G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.
48. V. Vipindeep and P. Jalote. List of common bugs and programming practices to avoid them. 2005.
49. J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.
50. A. C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS'82)*, 1982.