# A Hybrid Approach to
# Secure Function Evaluation using SGX

Joseph I. Choi
choijoseph007@ufl.edu
University of Florida

Dave (Jing) Tian
daveti@ufl.edu
University of Florida

Grant Hernandez
grant.hernandez@ufl.edu
University of Florida

Christopher Patton
cjpatton@ufl.edu
University of Florida

Benjamin Mood
bmood@pointloma.edu
Point Loma Nazarene University

Thomas Shrimpton
teshrim@ufl.edu
University of Florida

Kevin R. B. Butler
butler@ufl.edu
University of Florida

Patrick Traynor
traynor@ufl.edu
University of Florida

## ABSTRACT

A protocol for two-party secure function evaluation (2P-SFE) aims to allow the parties to learn the output of function $f$ of their private inputs, while leaking nothing more. In a sense, such a protocol realizes a trusted oracle that computes $f$ and returns the result to both parties. There have been tremendous strides in efficiency over the past ten years, yet 2P-SFE protocols remain impractical for most real-time, online computations, particularly on modestly provisioned devices. Intel's Software Guard Extensions (SGX) provides hardware-protected execution environments, called *enclaves*, that may be viewed as trusted computation oracles. While SGX provides native CPU speed for secure computation, previous side-channel and micro-architecture attacks have demonstrated how security guarantees of enclaves can be compromised.

In this paper, we explore a balanced approach to 2P-SFE on SGX-enabled processors by constructing a protocol for evaluating $f$ relative to a partitioning of $f$. This approach alleviates the burden of trust on the enclave by allowing the protocol designer to choose which components should be evaluated within the enclave, and which via standard cryptographic techniques. We describe SGX-enabled SFE protocols (modeling the enclave as an oracle), and formalize the strongest-possible notion of 2P-SFE for our setting. We prove our protocol meets this notion when properly realized. We implement the protocol and apply it to two practical problems: privacy-preserving queries to a database, and a version of Dijkstra's algorithm for privacy-preserving navigation. Our evaluation shows that our SGX-enabled SFE scheme enjoys a 38x increase in performance over garbled-circuit-based SFE. Finally, we justify modeling of the enclave as an oracle by implementing protections against known side-channels.

## CCS CONCEPTS

• **Security and privacy** → **Formal security models**; **Privacy-preserving protocols**; **Hardware-based security protocols**.

## KEYWORDS

secure function evaluation; SGX; partitioning; protocols

## 1 INTRODUCTION

Secure function evaluation (SFE) describes the process of multiple parties collectively computing a function and receiving its output without learning the inputs from any other party. Originally proposed in the 1980s, SFE was primarily of theoretical interest until the 2000s, when practical implementations of two-party SFE (2P-SFE) became available. Since then, interest in the space has dramatically increased and the costs of computation have been lowered by orders of magnitude.

Despite this success in reducing the costs of SFE, it is still not yet sufficiently practical to be used in applications where (near) real-time performance is required. This is in large part due to the substantial number of cryptographic operations that the parties need to perform. In the 2P-SFE case, this is often manifested by representing the function to be computed over as a circuit and *garbling* all of its input and output wires, as well as truth tables associated with each logic gate.

Hardware support for secure computing offers a chance to reduce these costs. Specifically, Intel's Software Guard Extensions (SGX) provides secure memory regions (called enclaves) inside which code and data can live outside of the purview of the operating system or system administrator. This platform thus offers the potential to enable SFE without the often crippling overheads of the associated cryptographic constructions. While SGX provides native CPU speed for secure computation, previous side-channel

and micro-architecture attacks have demonstrated how security guarantees of enclaves can be compromised. Controlled-channel attacks [68] leverage the page fault handler to leak sensitive information inside an enclave. Leaky Cauldron [64] shows memory side channel hazards in SGX ranging from TLB to DRAM modules. Meltdown [41] and Spectre [34] attacks can also be applied to enclaves, and Foreshadow [63] has successfully extracted the CPU attestation key from enclaves thus breaking the SGX remote attestation. Some of these attacks could be mitigated by microcode update or system hardening. However, they do expose two important questions:

(1) *Is it reasonable to put all secrets into an enclave?*
(2) *What could we do if SGX might be compromised?*

This paper goes beyond simply introducing SGX to SFE [5]. Rather, in addition to reducing the computation required for 2P-SFE operations, our scheme provides provable assurance that the most sensitive inputs of either party will remain protected. In summary, our main contributions[1] are as follows:

- **Design of Hybrid SFE Scheme:** We provide a new construction that considers the partitioning of a function into multiple segments, some to be executed within an SGX enclave and the others within a garbled circuit. We surface the idea of partitioning a function $f$ as a first-class primitive in the design, as different partitionings induce different schemes, with different efficiency and trust assumptions.
- **Formal Protocol Analysis:** We formalize two-party protocols with an explicit oracle, to support analysis of SFE protocols that leverage an SGX enclave. We also formalize a best-possible notion of 2P-SFE for our setting — traditional 2P-SFE is not possible, in general — and prove that our Hybrid SFE scheme, properly realized, meets this notion.
- **Evaluation and Case Studies:** We provide practical scenarios for using our SFE schemes, including a database interacting with a client for privacy-preserving queries and a location-finding scenario with a privacy-preserving Dijkstra's algorithm. Our extensive evaluation demonstrates the overhead of these operations and shows that in the hybrid SFE scheme, we increase performance by up to 38x compared to garbled circuits. We also provide an empirical demonstration of resilience to controlled-channel attacks.

OUTLINE. The rest of the paper proceeds as follows: Section 2 provides a high-level view of our hybrid approach; Section 3 provides preliminary notation; Section 4 provides a detailed formal treatment of SGX-enabled SFE; Section 5 defines and analyzes our hybrid protocol; Section 6 discusses side-channels and mitigations for our implementation; Section 7 presents the design and implementation of the protocol schemes within a real SGX environment; Section 8 describes our experimental evaluation; Section 9 considers related work; and Section 10 concludes.

## 2 HYBRID APPROACH HIGH-LEVEL VIEW

SGX provides a very efficient platform for secure computation. However, the trust model associated with SGX is substantially different than with garbled circuits or other cryptographic approaches
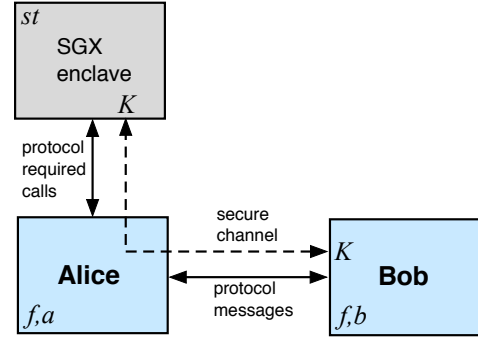


**Figure 1: High-level view of our execution model.**

to SFE. The secure enclave in which the computation is being performed must attest to the remote party providing data that it is trustworthy. To do this, the enclave must be issued an identifier by Intel and must use an attestation service provisioned by Intel. Unfortunately, Foreshadow attacks have demonstrated that it is hard to guarantee that the sensitive code and data are running in a real enclave rather than an emulation environment even if SGX remote attestation succeeds, leaving alone threats from other side-channel and micro-architecture attacks. Furthermore, the size of the enclave is limited by the enclave page cache, which is limited to approximately 128 MB.[2] For these reasons, parties may not want to rely solely on SGX to compute a function $f$ on their private inputs. However, traditional approaches to SFE using garbled circuits, even using the most efficient schemes, are still orders of magnitude slower than performing the same operations non-securely.[3]

We consider the middle ground: parts of the computation of $f$ are performed using SGX, and parts are performed using traditional SFE mechanisms. Exactly *how* one partitions the function is an efficiency- and security-critical matter. The efficiency viewpoint has already been touched upon, so we take up the security viewpoint.

Our setting is loosely captured in Figure 1. Alice and Bob would like to carry out a protocol for 2P-SFE of $f(a, b)$. Alice has black-box access to an SGX enclave that her environment hosts. (We will justify this black-box modeling in a moment.) Bob, the remote party, shares a secret key with the enclave, used to establish a secure communication channel; Alice has view of this channel, but does not possess the key. As Alice and Bob carry out the protocol, Alice will make "queries" to her enclave, asking it to compute intermediate functions that are specified by a given partitioning of $f$. She provides private input (related to $a$) as part of these queries, and Bob provides private input (related to $b$) via the secure channel. The results of these enclave queries are visible to Alice, and used in subsequent parts of the protocol for computing $f(a, b)$.

Thus, intermediate information about the computation of $f$ is leaked to Alice, even when she participates honestly. This implies that, in general, the standard notion of 2P-SFE *is not possible* in this setting. A bad partitioning of $f$ may result in intermediate values that leak information about private inputs $a$ and $b$ of Alice and Bob.

We define, and aim to achieve, the best possible SFE notion in this setting. Namely, to show that a protocol for computing $f(a, b)$ leaks nothing more than $f(a, b)$ and the intermediate values. That is, 2P-SFE with respect to a *particular way of partitioning the function.*

We formally model an SGX enclave as a black-box, but in reality, an enclave may leak information about the computations it performs due to timing side channels, memory access patterns, and controlled-channel attacks on program flow [68]. Enclave malware was recently demonstrated to be capable of cache attacks on co-located enclaves, recovering nearly entire RSA private keys within 5 minutes [53]. To support our black-box modeling of an enclave, we include mitigations for side-channels in our implementations.

## 3 PRELIMINARIES

In order to design a secure hybrid scheme using SGX, we first introduce the fundamental protocols and their associated notations. This includes our main protocol involving two parties Alice and Bob, an introduction to Garbled Circuit (GC) syntax, and finally *1-2 oblivious transfer* in the context of this work.

Let $\mathrm{dom} f$ denote the domain of a function $f$. We use $\varepsilon$ to denote the empty string. If $a$ and $b$ are strings, let $a \| b$ denote their concatenation. Let $y \leftarrow^\$ A(x_1, \ldots)$ denote the execution of a randomized algorithm $A$ on input $(x_1, \ldots)$ and assigning of the output to $y$. We write $y \leftarrow A(x_1, \ldots)$ if $A$ is deterministic. Algorithms are randomized unless noted otherwise.

### 3.1 Protocols

An *oracle-relative, two-party protocol* $\Pi$ is a two-party protocol played by Alice and Bob, in which one or both may have access to an explicitly defined oracle $\mathcal{O}$. The parties and the oracle all have local, private state, which includes a *long-term input* that is determined during protocol initialization and accessible across protocol executions. In our SGX-enabled protocols, the oracle abstracts an SGX enclave that is part of Alice's environment, and the long-term inputs of Bob and the oracle encode a shared key.

Protocols are executed with respect to players' private inputs. Executing the protocol on $(a, b)$ means to initiate Alice's state with private input $a$ and Bob's state with private input $b$, and exchange messages until both players halt. (We assume each player and the oracle have been provisioned with their long-term inputs.) This is denoted $(y_0, y_1, \pi, st') \leftarrow^\$ \Pi(1^k, a, b, st)$ where $st$ is the initial state of $\mathcal{O}$, $y_0$ and $y_1$ are the final states of Alice and Bob respectively, and $st'$ is the final state of $\mathcal{O}$. String $\pi$ is a "transcript" of the protocol execution.[4] $\mathrm{Out}^i_{\Pi, k}(a, b, st)$ is the random variable denoting the final state of player $i$ when executing protocol $\Pi$ on $(a, b)$ with $st$ as the oracle's initial state. (Integer $k$ is the security parameter.) Refer to Appendix A.1 for more details.

Note that our protocol syntax admits trivially secure SFE protocols. For us, the oracle provides a way to abstractly capture an

SGX enclave, and our formalization provides a convenient way to reason about SGX-enabled SFE protocols.[5]

ADVERSARIAL MODEL. We will consider the security of oracle protocols in the presence of *semi-honest* (sometimes called *honest-but-curious*) adversaries. This means each player executes the protocol faithfully, but may otherwise act arbitrarily to violate security. All of our notions ask the adversary to distinguish its view of the protocol from the output of a simulator, which is given the private input of the corresponding player (and only the length of the other player's private input). Although a *semi-honest* model may not be sufficient for certain classes of real-world applications, it offers a natural first step towards SGX-enabled SFE. In many situations, we can expect computing parties to have a mutual interest in fulfilling the protocol correctly. Our model nevertheless does not prevent an adversary from attempting to break into the enclave.

### 3.2 Garbling schemes

We adopt the syntax of Bellare, Hoang, and Rogaway [9] for garbled circuits. A *garbling scheme* is a quadruple of algorithms $\mathcal{G} = (\mathrm{Gb}, \mathrm{En}, \mathrm{De}, \mathrm{Ev})$; the first is randomized, while the rest are deterministic. The *garbling algorithm* Gb takes as input $1^k$ and a function $f$, and outputs a triple of strings $(F, e, d)$. This is written $(F, e, d) \leftarrow^\$ \mathrm{Gb}(1^k, f)$. String $e$ describes the *encoding function*, and $X \leftarrow \mathrm{En}(e, x)$ denotes the encoding of $x$ under $e$; we call $X$ the *garbled input*. String $F$ describes the *garbled function*, and $Y \leftarrow \mathrm{Ev}(F, X)$ denotes evaluating $F$ on $X$, yielding the *garbled output $Y$*. Finally, string $d$ describes the *decoding function*, and $y \leftarrow \mathrm{De}(d, Y)$ denotes the decoding of $Y$ under $d$, yielding the *final output $y$*. The garbling scheme $\mathcal{G}$ is *correct* if for every function $f$, for every $x$ in the domain of $f$, and for every $(F, e, d)$ in the range of $\mathrm{Gb}(1^k, f)$, it holds that $f(x) = \mathrm{De}(d, \mathrm{Ev}(F, \mathrm{En}(e, x)))$.

Garbling circuits were introduced by Andrew Yao [69]. His construction, as well as most recent designs, have an additional syntactic property necessary for SFE. A garbling scheme is called *projective* if the encoding algorithm may be written as a pair of algorithms $(\mathrm{EnA}, \mathrm{EnB})$ such that $\mathrm{En}(e, (a, b)) = \mathrm{EnA}(e, a) \| \mathrm{EnB}(e, b)$ for every $(a, b) \in \mathrm{dom} f$. (See Appendix A.2 for a more formal definition.)

Bellare, Hoang, and Rogaway [9] formalize a security notion for garbling schemes, which we will use here. The *privacy* of a garbling scheme captures the adversary's ability to discern anything about $f$ or $x$ given only $F$, $X$, and $d$. The notion is parameterized by *side information* about the function $f$ — for example, the length of its encoding, or the topology of the circuit used to compute it. The adversary is given the side information as input. In the SFE setting, the side information is $f$ itself. We formalize the simulation-based privacy notion of [9] in Appendix A.2.

### 3.3 1-2 oblivious transfer

A standard way of facilitating secure multiparty computation (and achieving SFE in particular) is to compose a projective garbling scheme with an oblivious transfer protocol [49]. A 1-2 (one-of-two) transfer protocol is a two-party protocol in which Alice possesses two equal-length strings $X^0$ and $X^1$ and Bob possesses a

---

[4] The transcript serves no functional purpose, so we omit the details here.

[5] Although we choose to reason about SGX-enabled SFE protocols with access to an oracle, it is also possible to prove composition for protocols without oracles.
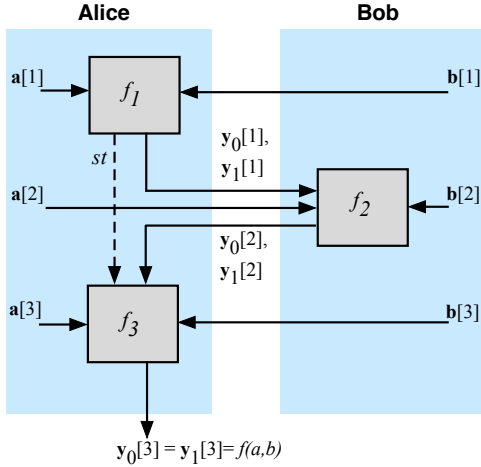
**Figure 2: A 3-way even-odd partitioning of function $f$. Solid lines are observed; dashed lines are not. Note that partitioning describes an organizational structure for computing $f$, but not *how* these computations are realized. In our protocols, $f_1, f_3$ will be computed within an SGX enclave, and $f_2$ via garbling schemes and oblivious transfer.**

bit $b$. Roughly speaking, a 1-2 transfer protocol is *oblivious* if Bob learns $X^b$ (and only $X^b$) and Alice learns nothing. More generally, Alice may possess a sequence of strings $(X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$ and Bob a string $b$ of length $n$, and the goal is to obliviously transfer $(X_1^{b_1}, \ldots, X_n^{b_n})$ to Bob. We formalize this security property in Appendix A.3.

## 4 SFE FOR PARTITIONED FUNCTIONS

We next describe our construction of SFE for partitioned functions.

Our main protocol partitions the computation of a function into a sequence of round functions, each depending on a piece of the players' private input. Odd-round functions are evaluated within Alice's enclave and may depend on its state. Even-round functions are stateless and evaluated using standard cryptographic techniques.

We first give the syntax of an partitioning scheme that captures this computational model.[6] We then define secure evaluation of a function relative to a partitioning of it, which captures the intermediate results available to Alice and Bob.

### 4.1 $\ell$-way even-odd partitioning schemes

Let $\ell$ be a positive integer and $f$ be a function of two inputs and two outputs. An *$\ell$-way even-odd partitioning scheme* $\mathcal{P}$ for $f$ is a sequence of round functions $(f_1, \ldots, f_\ell)$ and a pair of probabilistic algorithms (SpA, SpB). Let $(a, b) \in \text{dom} f$ and $k$ be a positive integer. On input $(1^k, a)$, algorithm SpA outputs an $\ell$-vector of strings $\mathbf{a}$. Similarly, on input $(1^k, b)$, algorithm SpB outputs an $\ell$-vector of

---

[6]Our aim is to demonstrate the possibility of combining SGX and Garbled Circuits for SFE, given a proper partitioning scheme. Partitioning is itself a hard problem that we do not attempt to solve; entire papers [17, 70] have been dedicated to it. Four possible partitioning schemes for SGX programs are explored by Atamli-Reineh and Martin [3], ranging from the naïve placement of entire applications in an enclave to separating out sensitive components into individual enclaves.

---

$$
\begin{array}{l}
\hline
\text{Exec}_{\mathcal{P}, k}(a, b) \\
\hline
\mathbf{a} \leftarrow_\$ \text{SpA}(1^k, a); \mathbf{b} \leftarrow_\$ \text{SpB}(1^k, b); st, \mathbf{y}_0[0], \mathbf{y}_1[0] \leftarrow \varepsilon \\
\text{for } j \leftarrow 1 \text{ to } \ell \text{ do} \\
\quad u \leftarrow \mathbf{a}[j] \parallel \mathbf{y}_0[j-1]; v \leftarrow \mathbf{b}[j] \parallel \mathbf{y}_1[j-1] \\
\quad \text{if } j \text{ is odd then } (\mathbf{y}_0[j], \mathbf{y}_1[j], st) \leftarrow f_j(u, v, st) \\
\quad \text{else } (\mathbf{y}_0[j], \mathbf{y}_1[j]) \leftarrow f_j(u, v) \\
\text{return } (\mathbf{a}, \mathbf{b}, \mathbf{y}_0, \mathbf{y}_1) \\
\hline
\end{array}
$$

**Figure 3: Execution of partitioning scheme $\mathcal{P}$ on input $(a, b)$. Note that prior outputs are concatenated to the inputs of the next round function.**

strings $\mathbf{b}$. These are the local inputs of Alice and Bob, respectively. Even-round function evaluations are stateless, while odd-round function evaluations carry state from one odd-round to the next. This allows us to model stateful SGX computation. Even-numbered functions map two strings (Alice and Bob's local inputs) to two strings (Alice and Bob's outputs), and odd-numbered functions map three strings (Alice and Bob's local inputs, and the state of the oracle) to three strings (Alice and Bob's outputs and the oracle's updated state).

We define the execution of $\mathcal{P}$ in Figure 3. (See Figure 2 for an illustration.) The partitioning scheme $\mathcal{P}$ is correct for $f$ if for every positive integer $k$ and every $(a, b) \in \text{dom} f$, it holds that

$$
\Pr\left[(\mathbf{a}, \mathbf{b}, \mathbf{y}_0, \mathbf{y}_1) \leftarrow_\$ \text{Exec}_{\mathcal{P}, k}(a, b) : (\mathbf{y}_0[\ell], \mathbf{y}_1[\ell]) = f(a, b)\right] = 1.
$$

We define $\mathcal{I}$ as the 1-way even-odd partitioning scheme defined for every function $f$ as follows. Let $(a, b) \in \text{dom} f$ and $k$ be a positive integer. On input $(1^k, a)$, algorithm $\mathcal{I}.$SpA outputs a 1-vector containing $a$, and on input $(1^k, b)$, algorithm $\mathcal{I}.$SpB outputs a 1-vector containing $b$. Correctness demands that $\mathcal{I}.f_1 = f$. We call $\mathcal{I}$ the *identity partition*.

### 4.2 Partition-relative (2P)-SFE

Let $f$ be a function of two inputs and two outputs and let $\mathcal{P}$ be a partitioning scheme for $f$. Syntactically, a protocol $\Pi$ for evaluating $f$ relative to $\mathcal{P}$ is an oracle protocol played by Alice and Bob with the following correctness condition: for each $i \in \{\text{Alice}, \text{Bob}\}$, every positive integer $k$, and every $(a, b) \in \text{dom} f$, where $(y_{\text{Alice}}, y_{\text{Bob}}) = f(a, b)$, it holds that

$$
\Pr\left[\mathbf{a} \leftarrow_\$ \mathcal{P}.\text{SpA}(1^k, a); \mathbf{b} \leftarrow_\$ \mathcal{P}.\text{SpB}(1^k, b) : \right.
$$

$$
\left. \text{Out}_{\Pi, k}^i(\mathbf{a}, \mathbf{b}, \varepsilon) = y_i\right] = 1,
$$

DEFINING SFE. Our base notion of SFE will be stated relative to a given partitioning scheme and admits the traditional notion of SFE as a special case. We formalize the idea that a protocol securely evaluates a function (relative to a partitioning of that function) if nothing is leaked to either party beyond that which follows from its own local inputs and outputs.[7] To do so, we insist that each party's *view* of the protocol's execution can be computed *without interacting with the other party*.

---

[7]If Alice's or Bob's private input is leaked in the intermediate values, this is a failure of the partitioning, not of the SFE protocol.

Let $f$ be a function, $\mathcal{P}$ be a partitioning scheme for $f$, and $\Pi$ be an oracle protocol for evaluating $f$ relative to $\mathcal{P}$. Security is captured by the game defined in the top-left panel of Figure 9 (located in the Appendix). We sketch the notion here. The security experiment is associated to player $i \in \{\text{Alice}, \text{Bob}\}$, adversary $\mathcal{A}$, simulator $\mathcal{S}$, and security parameter $k$. The goal of the adversary is to distinguish the view of player $i$ from the output of $\mathcal{S}$, which is given $f$, player $i$'s local inputs and outputs, and only the *length* of the other player's inputs and outputs. To begin, the adversary chooses a pair of inputs $(a, b) \in \text{dom} f$ and the inputs are split according to $\mathcal{P}$. A challenge bit $c$ is chosen. If $c = 1$, then the protocol is executed and the adversary is handed $i$'s view of the protocol execution, derived from the transcript; if $c = 0$, then the simulator is executed and the adversary is given its output. The adversary wins if it outputs $c$. The advantage of $\mathcal{A}$ in the game instantiated with simulator $\mathcal{S}$ at security parameter $k$ is defined as

$$\text{Adv}^{\text{sfe}}_{\Pi, \mathcal{P}, f, i}(\mathcal{A}, \mathcal{S}, k) = 2 \cdot \text{Pr}\left[ \text{Exp}^{\text{sfe}}_{\Pi, \mathcal{P}, f, i}(\mathcal{A}, \mathcal{S}, k) = 1 \right] - 1.$$

We say that $\Pi$ *securely evaluates* $f$ *relative to* $\mathcal{P}$ if for each $i \in \{\text{Alice}, \text{Bob}\}$ and every polynomial-time $\mathcal{A}$, there exists a polynomial-time $\mathcal{S}$ such that $\text{Adv}^{\text{sfe}}_{\Pi, \mathcal{P}, f, i}(\mathcal{A}, \mathcal{S}, k)$ is negligible as a function of the security parameter $k$. We refer to such a protocol as an SFE-secure protocol for $f$ relative to $\mathcal{P}$.

When $\mathcal{P} = \mathcal{I}$, our notion reduces to standard SFE, and we simply say that $\Pi$ *securely evaluates* $f$, dropping the reference to the partitioning.

## 5 HYBRID SFE-SGX

Now having discussed our model for Secure Function Evaluation (SFE) on a partitioned function, we extend this to the context of computing using Intel SGX. It is critical that we define our own syntax and construction, despite formalisms already existing [5, 59], as there is no previous work that partitions Secure Multiparty Computation between SGX hardware and traditional Garbled Circuits.

Let $\mathcal{P}$ be an $\ell$-way even-odd partitioning of $f$, where $f_1, \ldots, f_\ell$ are the round functions. The main technical result of this section says, loosely, that if one possesses protocols for securely evaluating $f_1, \ldots, f_\ell$, then these can be composed to give a protocol for securely evaluating $f$ relative to the information leaked by the component functions. This result highlights the need to carefully consider how $f$, and the private inputs $a, b$, are partitioned. Concretely, one may have a protocol for securely computing $f_1(a_1, b_1, st)$, which leaks nothing more about $a_1, b_1, st$ than $f_1$ does itself. But $f_1$ may, for example, leak all of $a_1$. The claimed implication will hold, but if $a_1 = a$, there is no security in the classical sense of SFE.[8]

Before we can state our main result, we define (as technical tools) syntax and security notions for protocols for even- and odd-round function evaluations.

EVEN-ROUND PROTOCOLS. An even-round protocol $\Pi$ is an oracle protocol played by Alice and Bob for evaluating functions of two inputs and two outputs. We write $\Pi[f]$ to denote the protocol instantiated with a particular function $f$. Correctness demands

---

[8]We note that this particular leakage is ameliorated if $a_1$ itself leaks no efficiently computable information about $a$, e.g., $a_1$ is an encryption of $a$ under a secret key.

---

$\Pi_{\text{hyb}}[\mathcal{P}, \Pi_{\text{o}}, \Pi_{\text{e}}](1^k, \mathbf{a}, \mathbf{b})$

(1) Let $st, \mathbf{y}_0[0], \mathbf{y}_1[0] \leftarrow \varepsilon$. For each $j$ from 1 to $\ell$, do as follows: let $u = \mathbf{a}[j] \, \| \, \mathbf{y}_0[j-1]$ and $v = \mathbf{b}[j] \, \| \, \mathbf{y}_1[j-1]$. If $j$ is odd, then execute

$\quad (\mathbf{y}_0[j], \mathbf{y}_1[j], \pi, st) \leftarrow^\$ \Pi_{\text{o}}[\mathcal{P}.f_j](1^k, u, v, st)$;

otherwise, execute

$\quad (\mathbf{y}_0[j], \mathbf{y}_1[j], \pi, st') \leftarrow^\$ \Pi_{\text{e}}[\mathcal{P}.f_j](1^k, u, v, \varepsilon)$.

(2) Alice halts with private state $\mathbf{y}_0[\ell]$ and Bob halts with private state $\mathbf{y}_1[\ell]$.

**Figure 4: The hybrid SFE-SGX protocol, an oracle protocol for evaluating $f$ constructed from $\ell$-way even-odd partitioning scheme $\mathcal{P}$ for $f$, odd-round protocol $\Pi_{\text{o}}$, and even-round protocol $\Pi_{\text{e}}$.**

that for each $i \in \{\text{Alice}, \text{Bob}\}$, every $(a, b) \in \text{dom} f$, and every positive integer $k$, it holds that $\text{Pr}[\text{Out}^i_{\Pi[f], k}(a, b) = y_i] = 1$ where $(y_{\text{Alice}}, y_{\text{Bob}}) = f(a, b)$. We consider the SFE security of even-round protocols relative to the identity partition.

ODD-ROUND PROTOCOLS. An odd-round protocol $\Pi$ is an oracle protocol played by Alice and Bob for evaluating functions of three inputs and three outputs. We write $\Pi[f]$ to denote the protocol instantiated with a particular function $f$. Correctness demands that for each $i \in \{\text{Alice}, \text{Bob}\}$, every $(a, b, st) \in \text{dom} f$, and every positive integer $k$, it holds that $\text{Pr}[\text{Out}^i_{\Pi[f], k}(a, b, st) = y_i] = 1$, where $(y_{\text{Alice}}, y_{\text{Bob}}, st') = f(a, b, st)$ for some $st' \in \{0, 1\}^*$.

DEFINING SFE-ODD. We introduce a new security notion for odd-round protocols. The main distinction from standard SFE is that the adversary specifies the oracle's initial state. Security of odd-round protocols is defined in the top-right panel of Figure 9 (located in the Appendix). We define the advantage of adversary $\mathcal{A}$ in the game instantiated with simulator $\mathcal{S}$ as

$$\text{Adv}^{\text{sfe-odd}}_{\Pi, f, i}(\mathcal{A}, \mathcal{S}, k) = 2 \cdot \text{Pr}\left[ \text{Exp}^{\text{sfe-odd}}_{\Pi, f, i}(\mathcal{A}, \mathcal{S}, k) = 1 \right] - 1.$$

We say that $\Pi$ is an SFE-ODD-secure protocol for $f$ if for each $i \in \{\text{Alice}, \text{Bob}\}$ and every polynomial-time adversary $\mathcal{A}$, there exists a polynomial-time simulator $\mathcal{S}$ such that the function $\text{Adv}^{\text{sfe-odd}}_{\Pi, f, i}(\mathcal{A}, \mathcal{S}, k)$ is negligible as a function of $f$.

### 5.1 The hybrid SFE-SGX protocol

Our main protocol is defined in Figure 4. Let $f$ be a function of two inputs and two outputs and let $\mathcal{P}$ be an $\ell$-way even-odd partitioning scheme for $f$. The protocol is composed from $\mathcal{P}$, an odd-round protocol $\Pi_{\text{o}}$, and an even-round protocol $\Pi_{\text{e}}$. The protocol is defined on $\ell$-vectors $\mathbf{a}$ and $\mathbf{b}$ corresponding to Alice and Bob's respective split inputs. They first execute $\Pi_{\text{o}}[\mathcal{P}.f_1]$ with private inputs $(\mathbf{a}[1], \mathbf{b}[1])$ with oracle $\Pi_{\text{o}}.\mathcal{O}$. As a result, Alice gets $\mathbf{y}_0[1]$, Bob gets $\mathbf{y}_1[1]$, and the oracle's state gets updated to $st$ where $(\mathbf{y}_0[1], \mathbf{y}_1[1], st) = \mathcal{P}.f_1(\mathbf{a}[1], \mathbf{b}[1], \varepsilon)$. Next, they execute $\Pi_{\text{e}}[\mathcal{P}.f_2]$ on private inputs $(\mathbf{a}[2] \, \| \, \mathbf{y}_0[1], \mathbf{b}[2] \, \| \, \mathbf{y}_1[1])$ and with oracle $\Pi_{\text{e}}.\mathcal{O}$. Alice gets $\mathbf{y}_0[2]$ and Bob gets $\mathbf{y}_1[2]$ as a result. Alice and Bob continue in this way, alternating between odd-round and even-round evaluations. Correctness of $\mathcal{P}$ ensures that $(\mathbf{y}_0[\ell], \mathbf{y}_1[\ell]) = f(a, b)$.

$$\underline{\Pi_{gc}[f, \mathcal{G}, \Pi_{ot}](1^k, a, b)}$$

$\underline{\text{Init}(1^k)}$: return $(\varepsilon, \varepsilon, \varepsilon)$

(1) Alice generates the circuit $(F, e, d) \leftarrow\!\!\$ \text{ Gb}(1^k, f)$, computes
    $A \leftarrow \text{EnA}(e, a)$, and sends $(F, A)$ to Bob.
(2) Let $e = (X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$. Execute
    $$(\varepsilon, B, \pi, st') \leftarrow\!\!\$ \Pi_{ot}(1^k, e, b, \varepsilon).$$
    (Note that $B = \text{EnB}(e, b)$.)
(3) Bob computes $Y \leftarrow \text{Ev}(F, A \parallel B)$ and sends $Y$ to Alice.
(4) Alice computes $y \leftarrow \text{De}(d, Y)$.
(5) Alice halts on $y$ and Bob halts on $\varepsilon$.

**Figure 5: An even-round protocol constructed from a projective garbling scheme** $\mathcal{G} = (\text{Gb}, \text{En}, \text{Ev}, \text{De})$ **and 1-2 transfer protocol** $\Pi_{ot}$. **See Appendix A.1 for the semantics of** $\text{Init}$.

$$\underline{\Pi_{sgx}[f, \Gamma](1^k, a, b, st)}$$

$\underline{\text{Init}(1^k)}$: $K \leftarrow\!\!\$ \mathcal{K}$; return $(\varepsilon, K, K)$

(1) Bob computes $c \leftarrow\!\!\$ \mathcal{E}_K(b)$ and sends $c$ to Alice.
(2) Alice asks $(f, a, c)$ of $\mathcal{O}(K, \cdot)$.
(3) On input $(K, (f, a, c))$ and with current state $st$, oracle $\mathcal{O}$
    computes $b \leftarrow \mathcal{D}_K(c)$, $(y, st) \leftarrow f(a, b, st)$, and returns $y$ to
    Alice.
(4) Alice halts on $y$ and Bob halts on $\varepsilon$.

**Figure 6: A odd-round protocol constructed from a symmetric encryption scheme** $\Gamma = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ **and the SGX module, modeled as an oracle queried by** Alice. **See Appendix A.1 for the semantics of** $\text{Init}$.

We instantiate the even-round protocol using standard techniques from GC-based SFE and specify an example, constructed from a projective garbling scheme and a 1-2 transfer protocol, in Figure 5. This protocol does not make use of an oracle, relying only on the security of its constituent cryptographic primitives.

Finally, we instantiate the odd-round protocol using the SGX module (modeled as an oracle) and a symmetric encryption scheme $\Gamma = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. (SGX uses AES-GCM with a random initialization vector. See Appendix A.4 for syntax and security notions for symmetric encryption.) The protocol is defined in Figure 6. Bob shares a key with Alice's enclave. To evaluate a function $f$ on $(a, b)$, Bob encrypts $b$ and sends it to Alice, who then sends the function, her own input, and Bob's ciphertext to the SGX module. The module decrypts $b$, evaluates the function (which depends on its internal state), then returns the result to Alice.

The hybrid SFE-SGX protocol instantiated with $\Pi_{sgx}$ and $\Pi_{gc}$ is useful for evaluating functions whereby Alice gets the result and Bob gets nothing. More precisely, $\Pi_{hyb}[\mathcal{P}, \Pi_{sgx}, \Pi_{gc}]$ is well-defined when $\mathcal{P}$ is an even-odd partitioning of a function $f$ such that for every $(a, b) \in \text{dom} f$, it holds that $f(a, b) = (y, \varepsilon)$ for some string $y$. This is not always desirable; in some applications, Bob should receive the final result. To address this, we specify the *dual* protocol of hybrid SFE-SGX, whereby Bob learns all of the intermediate results, including the final result, and Alice learns nothing.

## 5.2 The dual hybrid SFE-SGX protocol

Let $f$ be a function of two outputs and two inputs such that for every $(a, b) \in \text{dom} f$, there exists a string $y$ such that $f(a, b) = (\varepsilon, y)$. Let $\mathcal{P}$ be an even-odd partitioning scheme for $f$. Protocol $\Pi'_{gc}$ is like its dual $\Pi_{gc}$, except that Alice also sends Bob the string $d$, the means to decode, in step 1. In step 3, Bob computes $Y \leftarrow \text{Ev}(F, A \parallel B)$ and decodes $y \leftarrow \text{De}(d, Y)$ and halts on $y$. Alice halts on output $\varepsilon$. Protocol $\Pi'_{sgx}$ is like its dual $\Pi_{sgx}$, except that in step 3, after computing $(y, st) \leftarrow f(a, b, st)$, the oracle encrypts $y$ under $K$ and returns the ciphertext to Alice. Alice transmits the ciphertext to Bob and halts on $\varepsilon$. Finally, Bob decrypts and halts on $y$.

Then $\Pi_{hyb}[\mathcal{P}, \Pi'_{sgx}, \Pi'_{gc}]$ is, syntactically, a protocol for evaluating $f$ relative to $\mathcal{P}$. Unlike its dual, Bob receives the final and intermediate outputs and Alice receives nothing.[9]

## 5.3 Case Study: Database

Having exhaustively defined the necessary partitioning scheme in developing a hybrid model for SGX, we present two case studies to demonstrate real-world use of our protocols.

The first of these involves querying a database. Alice provides a database with $n$ rows, and Bob issues $k$ SELECT queries against it. Each query retrieves a single 64-bit entry associated with the index provided as input. The results of the queries are returned to Bob.[10] The goal in this setting is to provide the requested entries to Bob without revealing any queried indices to Alice. Using our hybrid protocol, Bob would protect his most sensitive queries using GC-based computation and the less sensitive ones with SGX-enabled computation. Such a scenario may be practical in the case of a database implementing a simple Multilevel Security (MLS) scheme, where data is encoded at one of two levels (i.e., secret, top secret). This approach would allow the existence of queries to *potentially* (but not necessarily) top secret data to be obfuscated, without incurring the expense of using garbled circuits for all accesses.

HYBRID PROGRAM. This program can be evaluated using the dual hybrid SFE-SGX protocol. Some portion of the queries entered by Bob are considered highly sensitive, meaning Bob is especially concerned with not having them revealed. Bob splits his $k$ queries into two bins corresponding to $f_1$, the SGX portion of the program, and $f_2$, the GC portion. Alice loads the database into her enclave, which will evaluate $f_1$, and Bob issues queries against the enclave from his bin of less-sensitive inputs. Once the requested entries are returned to Bob, the two parties switch from $f_1$ to $f_2$ by having Alice's enclave pass the database into the garbled circuit as $y_1$, the garbled version of the database. Bob garbles the inputs corresponding to his highly sensitive queries by performing 1-2 OT with Alice. Bob then evaluates the garbled circuit and receives the requested entries associated with his more sensitive inputs. Alice has no output.

---

[9]Some applications may require both Alice and Bob to receive intermediate and final outputs. We note that our hybrid SFE-SGX and dual hybrid SFE-SGX protocols would behave like traditional GC if both parties are given the final output, but intermediate outputs must be handled more carefully. Intermediate outputs based from odd-rounds may leak information about the more-sensitive data handled by even-rounds. This is a line of inquiry beyond the scope of this paper, but it is certainly important.

[10]Although the database program we architect resembles a simple key-value store, it offers an initial demonstration that database retrieval is possible according to our hybrid scheme. No work had previously attempted to combine a database application across SGX and GC. Our database supports both retrieval and updates to data entries in a manner which corresponds to GET and SET requests against production databases.

## Dijkstra Hybrid

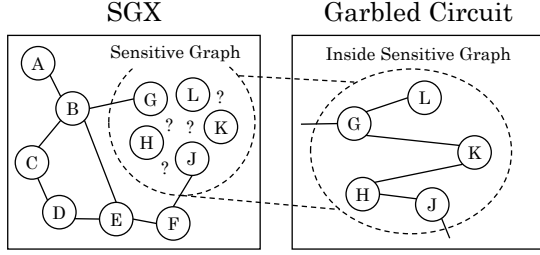### SGX        Garbled Circuit



**Figure 7: A hybrid Dijkstra graph is partitioned into two parts. The majority of the shortest path is found by the hybrid SFE-SGX computation; the sensitive part of the route is computed using garbled circuits.**

INTUITION FOR FURTHER ROUNDS. If the output Bob receives in the second round above advises the next set of data queries, the program's evaluation may be extended into further rounds. At this stage, Alice does not know Bob's GC output from the prior round, but any future queries, if dependent on this intermediate output, may leak information to Alice about Bob's sensitive inputs. If there are many result sets which could have advised Bob's next set of queries (or if $f_2$ was empty), Bob may continue as before, splitting his queries into two bins corresponding to $f_1$ and $f_2$, participating in another series of odd- and even-rounds. Otherwise (if Bob's next set of queries could only be a result of a few specific outputs from the initial GC round), all subsequent queries must be made exclusively in even-rounds (by emptying $f_1$ or filling it with dummy queries).

### 5.4 Case Study: Dijkstra's Shortest-Path

Our second case study uses Dijkstra's shortest-path algorithm. This algorithm finds the shortest path between two nodes in a graph. Again, there are two parties, Alice and Bob. The graph topology is known to both Alice and Bob, but only Bob knows the edge weights. Bob provides as input the graph's edge weights, and Alice inputs the starting and ending points. The goal of SFE in this setting is for Alice to receive the shortest path from her starting point to her ending point while not learning Bob's edge weights.

HYBRID PROGRAM. This program is evaluated using the standard hybrid SFE-SGX protocol. Some subset of the nodes in the graph make up a highly-sensitive portion of the map (graph), and Bob is especially concerned with not having the edge weights incident to them revealed while routing through that portion of the map. The highly-sensitive portion will be routed solely in a garbled circuit during $f_2$, whereas the less-sensitive portion will be routed within the SGX module during $f_1$. The final route may contain nodes from both portions (Figure 7 demonstrates how a graph may contain both highly-sensitive and less-sensitive portions). Any route through the graph is allowed to travel into and out of the sensitive part of the graph a single time; this is possible by setting edge weights in a specific way. The path must not start or end in the highly sensitive part of the graph. This partitioning could be useful in a scenario where roads traverse private/government property with selective access, the topological features of which must remain undisclosed.

The initial Dijkstra computation is run as $f_1$ in Alice's enclave. Alice's input to $f_1$ is the starting and ending points, and Bob passes the less-sensitive edge weights into the enclave. A route through the non-sensitive portion of the graph is calculated. This route is returned to Alice from the enclave, though it only covers the non-sensitive portion of the graph and may not yet be the complete route. Afterward, a garbled circuit is used to route through the highly-sensitive portion of the graph; the garbled circuit is evaluated regardless of whether the path goes through the highly-sensitive portion of the graph. The entrance and exit nodes for the sensitive portion of the graph are provided by the enclave as $y_1$ to $f_2$. Alice has no input into $f_2$. Bob's input into $f_2$ is the collection of sensitive edge weights. Upon evaluation of the garbled circuit associated with $f_2$, Alice receives as output the route through the highly sensitive portion of the graph. Bob has no output.

INTUITION FOR FURTHER ROUNDS. At the completion of the first series of rounds, Alice knows a shortest path between her starting and ending points. This may be part of a larger path, in the case that there are certain nodes Alice must visit en-route. However, Alice's new starting and ending points will not be dependent on the prior output. More rounds may nevertheless be required if, for instance, the same node should not be revisited, in which case the modified graph (with visited nodes removed) will be fed directly into the next series of rounds while Alice provides her new starting and ending points (for continuing the path).

### 5.5 Security of hybrid SFE-SGX

Our main result is that the composition of an odd-round and an even-round protocol achieves secure function evaluation relative to a particular partitioning of the function. The following theorem says that as long as the evaluations of even-round functions are SFE-secure and the evaluations of odd-round functions are SFE-ODD-secure, then the hybrid SFE-SGX protocol securely evaluates function $f$ relative to even-odd partitioning scheme $\mathcal{P}$ for $f$. We discuss proofs of this section's theorems in our full paper [16].

THEOREM 5.1. *Let $f$ be a function of two inputs and two outputs and $\mathcal{P}$ be an $\ell$-way even-odd partitioning scheme for $f$, where $\ell$ is a positive integer. Let $\Pi_e$ be an even-round protocol and $\Pi_o$ be an odd-round protocol. Let $\Pi = \Pi_{\text{hyb}}[\mathcal{P}, \Pi_o, \Pi_e]$ as defined in Figure 4. If $\Pi_o[\mathcal{P}.f_j]$ is an SFE-ODD-secure protocol for evaluating $\mathcal{P}.f_j$ for every odd $j$ and $\Pi_e[\mathcal{P}.f_j]$ is an SFE-secure protocol for evaluating $\mathcal{P}.f_j$ for every even $j$, then $\Pi$ securely evaluates $f$ relative to $\mathcal{P}$.*

The following theorem is a standard result, which says that the composition of a private, projective garbling scheme and a 1-2 oblivious transfer protocol yields secure evaluation of the even-round functions.

THEOREM 5.2. *Let $f$ be a function of two inputs and two outputs, $\mathcal{G}$ be a projective garbling scheme, $\Pi_{\text{ot}}$ be a 1-2 transfer protocol, and let $\Pi = \Pi_{\text{gc}}[f, \mathcal{G}, \Pi_{\text{ot}}]$ as defined in Figure 5. If $\mathcal{G}$ is private and $\Pi_{\text{ot}}$ is oblivious, then $\Pi$ is an SFE-secure protocol for $f$.*

This result represents the best-case scenario in the setting where Alice is given the internal state of her SGX module. Since even-round evaluations remain secure, she learns nothing beyond that which follows from the result of each even-round computation and possession of the inputs Bob sent encrypted to the SGX module.

Lastly, the following theorem says that, viewing the SGX module as an oracle, secure symmetric encryption suffices for SFE-ODD-secure evaluation of the odd-round functions. (IND$ refers to the standard indistinguishability notion for encryption schemes defined in Appendix A.4.)

THEOREM 5.3. *Let $f$ be a function of three inputs and three outputs, $\Gamma = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, and let $\Pi = \Pi[f, \Gamma]$ as defined in Figure 6. If $\Gamma$ is IND$-secure, then $\Pi$ is SFE-ODD-secure for $f$.*

We remark that the protocol $\Pi_{\text{hyb}}$ instantiated with $\Pi'_{\text{gc}}$ and $\Pi'_{\text{sgx}}$ as the even- and odd-round protocols is also secure. The proof follows closely the justifications of Theorems 5.2 and 5.3.

In the next section, we justify modeling SGX as a black-box by addressing side-channel attacks. But first, we remark that even if these protections do not suffice, our protocol still provides a measure of assurance. Suppose, in the worst case, that Alice is given the internal state of her enclave (and hence any keys shared with Bob). Secure evaluation of $f(a, b)$ is out of reach in this setting, since Alice learns some of Bob's local inputs and outputs. Still, Bob's most sensitive inputs remain secure (even upon enclave compromise, as an enclave only handles private inputs for odd-rounds): Alice learns nothing about them beyond that which follows from knowing the state of her enclave and the result of the computation.

# 6 SIDE-CHANNEL ATTACK MITIGATIONS

Unfortunately, SGX enclaves suffer from side-channels which may leak data regarding program execution and data [28, 32, 57]. In order to continue evaluating our hybrid scheme, we must justify the oracle assumption on SGX heuristically by using code modifications to close known side channels.[11] These channels are associated with *timing* and *program flow*, which allow an outside observer to infer what path a program takes during its execution or what memory is being accessed (due to access times), and *memory accesses*, where an attacker observes the parts of RAM accessed by the program.

TIMING. We take steps to ensure our programs run for the same amount of time regardless of the input, underlying values in memory, or the result. We ensure both branches of every `if` statement take the same amount of time and fix loop bounds that could otherwise reveal how many times a loop executes [18]. An example of this would be to have each branch perform the same amount of work (e.g., a single assignment to a variable).

PROGRAM FLOW. Even though `if` statements take the same amount of time, controlled-channel attacks [68] on the program flow are still possible (i.e., if the `true` code branch causes a instruction page miss, then this leaks information). To further mitigate these, we modify all `if` statements to prevent branches. For example, `if(a == b) c = 1 else c = 2` can be converted into the branch-free version `int t = a == b; t = makeSameAsBit0(t), c = (t & 1) | ((¬t) & 2)`, where `makeSameAsBit0` sets all bits of `t` to the least-significant bit. In the new version, there is no branch that would reveal information about the program flow.[12]

MEMORY ACCESSES. Any query to a specific array index will reveal the memory page that was queried if that page faulted. Additionally, cache side channels can reveal to motivated attackers which data or code memory addresses are being accessed [12]. To counter this, we move RAM accesses from programs to a binary search tree (BST)-based Oblivious RAM (ORAM)-like construction loosely based on Path ORAM [60]. This modification distributes array accesses across many randomized memory accesses, as is visually apparent in Figure 8. The security model for our BST structure is similar to that of other ORAM systems, but is different in two important ways: (1) we deal with a trusted module in a compromised, malicious operating system [51] which may attempt timing-based attacks, and (2) our modifications are designed to hide program flow, not just data access patterns.[13]

Let us briefly summarize the design of our system.

- Our BST forest contains two trees, each with about half the nodes at any given time and in a sorted order, but no duplicate entries with the other. Each node has a key-value pair, where the key is array index and the value is the data.
- Every query for a node must descend through each tree all the way to a leaf. After the query, each node along the path is removed, moved to a different RAM location, and randomly added back into a tree. To hide where each key and data were moved, we place every deleted node's data to a contiguous segment of memory, then randomly swap them using the const-time and branch-free operations from before.
- To hide the actual query, $\log(n)$ queries are issued to each tree for every real query.

Whenever there is code that may reveal the searched index, we employ constant-time and branch-free techniques from the program flow section to hide the information.

# 7 DESIGN AND IMPLEMENTATION

In Section 5, we assumed Alice possessed an SGX module with which Bob shared a secret key. Here, and in our implementation, we assume for simplicity that both parties possess an SGX module. The two modules share a key unknown to either Alice or Bob, constructed during SGX remote attestation and used to set up a secure communication channel between the modules. We refer to the party who executes code in his/her enclave as the *evaluator*. The other party is called the *sender*.

We implement our protocols using Intel's SGX SDK for Linux [29]. Rather than simply replacing GC computation using SGX [5], we provide the first implementation that we are aware of combining SGX and GC together to perform 2P-SFE, which is essentially a general form of computation model for 2P-SFE to use SGX. Unlike previous work [5], we also share our experience of using Intel SDK with SGX-enabled hardware to develop real-world SFE applications.

PROGRAM PARTITIONS. When developing programs for use in the SGX environment, it is particularly important to understand the security considerations of code and data use. The first step is to

---

[11]For this work, our mitigations are applied manually for each function. The availability of a tool that automates this process would make it easier to adopt our hybrid protocol.
[12]Our approach differs from Raccoon [50], which executes multiple program paths on a given input; instead, we do away with branching paths altogether prior to execution.

[13]As opposed to traditional (e.g., Path) ORAM, we do not deal with a client/server setting, in which the server is not trusted to correctly perform the ORAM. The ORAM functionality within the enclave can be verified during attestation, making it unnecessary to maintain the position map and stash in a separate "trusted client."
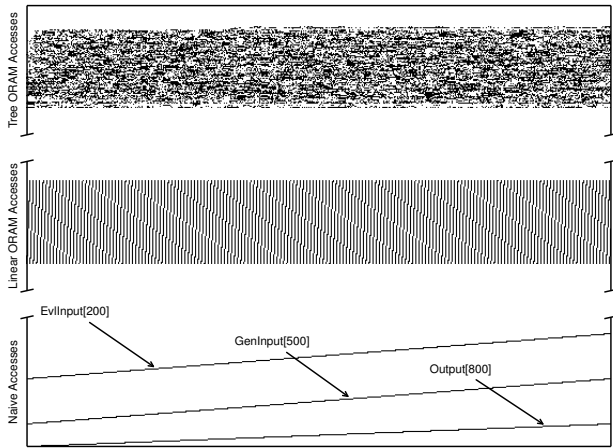
**Figure 8: A display of memory accesses of the database program. Top: tree-based ORAM; middle: linear search. The three lines on the bottom are unblinded memory accesses. We queried the memory in order from index 0 to $N$.**

partition the program into two parts. The *trusted* partition of the program is kept within the enclave, while the remainder is developed outside of the enclave. This program partitioning is required to minimize the application's trusted computing base and to save enclave memory. In our experience, the enclave's memory is fixed by a BIOS setting.[14] This represents a substantially different programming paradigm from what is required in other SGX-like environments such as the OpenSGX [31] emulator, where programs of any size are assumed able to fit entirely within the enclave environment.

All of our SGX test programs are executed inside of the evaluator's enclave. All other components, such as code to handle sockets and message processing, are kept outside the enclave. In total, $10,172$ SLOC[15] were written between the sender and evaluator enclaves and their accompanying untrusted applications.

Enclave Restrictions. Unlike OpenSGX, where enclave implementations can take advantage of existing libraries, real enclaves not only require library code to be statically linked, but also use *trusted libraries*. Static linking guarantees each enclave is self-contained and needs no extra libraries to be installed to where it is deployed. The trusted libraries created by Intel for enclave programs are crafted to avoid illegal[16] instructions (e.g., `fprintf`) that will crash enclave programs at runtime. Certain functions (e.g., `strcpy`) are excluded, though variants providing similar functionality (e.g., `strncpy`) are often available. Functions that would otherwise access data outside the enclave (e.g., `fopen`) are also excluded.

The Intel SGX SDK provides trusted C and C++ libraries, as well as other trusted libraries (e.g., for SGX runtimes and cryptographic operations). We follow these restrictions, primarily using a subset of C functions and some basic C++ built-in data structures (e.g., `vector`). When needed, we use the SGX SDK trusted libraries

for API replacements or alternate functions. For example, we use `sgx_read_rand`, which accesses the hardware random number generator directly, instead of `rand`. Similarly, we rely on the trusted libraries' cryptographic APIs instead of OpenSSL/GnuTLS.

SGX/SDK Restrictions. The SGX remote attestation protocol relies on Intel's Enhanced Privacy ID (EPID) [1] technology to verify that the remote enclave is running on an Intel-authenticated, SGX-enabled CPU. To use EPID, a signed certificate must first be obtained from a recognized certificate authority and registered with Intel Development Services. For the purposes of this work, we do not purchase a certificate for registration with Intel and do not implement the EPID part of remote attestation. For our experiments, we instead enable the debug flag when building our SGX applications to skip EPID. After skipping EPID, the sender in our setup verifies the measurement of the evaluator's enclave by checking the signed value contained in the SGX quote (from the attestation) against the known "good" quote of the enclave.

We encountered instances where we received errors from SGX when attempting to create and read data from large enclaves. Unlike normal programs, enclaves require explicit settings for both stack and heap size. Both also need to be 4K aligned (a normal page size). For memory-hungry enclaves, maximum stack and/or heap sizes must be increased in the enclave configuration file.[17]

Garbled Circuit Implementation. We used the Frigate semi-honest garbled circuit compiler and semi-honest protocol implementation provided by Mood et al. [45]. This implementation is a hand-tuned version of the garbled circuit system by Kreuter et al. [37]; it uses the point-and-permute [7], garbled row reduction [48], and free XOR [35] optimizations, amongst many others, to improve computational efficiency and reduce network bandwidth.

## 8 EXPERIMENTS

We used three common programs in the SFE literature to evaluate the performance of our implementation. We compared our hybrid SFE-SGX protocol to naïve SGX-enabled SFE (function evaluation in the enclave without side-channel protections), SGX-enabled SFE (with side-channel protections), and standard GC-based SFE.

We tested our implementation on two HP Envy360 laptops with Intel quad core i7-6500U CPUs at 2.50GHz with a 64KB L1 cache, 512KB L2 cache, 4MB L3 cache and 8 GB RAM. Both machines were connected on a VLAN to the same switch via Gigabit Ethernet.

### 8.1 Test Programs

The two programs used for evaluation of our hybrid SFE-SGX protocol were introduced in Section 5, being Database and Dijkstra's shortest-path. Additionally, we use the Millionaires Problem for evaluation of our naïve SGX-enabled SFE and SGX-enabled SFE protocols.[18] We now present configurations of each test program, with additional detail for hybrid versions of Database and Dijkstra.[19]

---

[14]In our environment, enclave memory was limited to 128 MB.

[15]Source Lines of Code (SLOC) generated using David A. Wheeler's 'SLOCCount.'

[16]We borrow this verbiage from the official Intel SGX documentation.

[17]The default maximum heap size (or total memory available for dynamic allocation) of an enclave is 1 MB. Any further allocation would either trigger an out-of-memory error if the application is in an ECall, or crash the enclave and application.

[18]We use the Millionaires Problem only to illustrate the differences in performance of our naïve SGX-enabled SFE/SGX-enabled SFE protocols and pure GC. We do not implement a hybrid version, but not due to its incompatibility with hybrid computation.

[19]In Section 5, we provided intuition for extending the hybrid versions of Database and Dijkstra into further rounds. That discussion was meant primarily to demonstrate

|  | | Time (ms) | | | |
|---|---|---|---|---|---|
| Program | Naïve | SGX-enabled SFE | | Hybrid | GC |
| Millionaires1024 | 113 ± 3% | 114 ± 2% | | - | 697 ± 2% |
| Millionaires4096 | 111 ± 2% | 110 ± 2% | | - | 1,640 ± 1% |
| Millionaires16384 | 116 ± 3% | 114 ± 3% | | - | 5,468 ± 0.2% |
| Millionaires262144 | 121 ± 2% | 121 ± 2% | | - | 82,960 ± 0.4% |
| Dijkstra20 | 112 ± 4% | 118 ± 4% | | 1,814 ± 0.2% | 1,086 ± 0.4% |
| Dijkstra50 | 111 ± 3% | 115 ± 3% | | 1,820 ± 0.2% | 4,788 ± 0.1% |
| Dijkstra100 | 112 ± 2% | 120 ± 3% | | 2,333 ± 0.2% | 20,023 ± 0.02% |
| Dijkstra200 | 117 ± 2% | 128 ± 1% | | 6,560 ± 0.2% | 78,905 ± 0.02% |
| Dijkstra250 | 119 ± 2% | 133 ± 1% | | 23,330 ± 0.2% | 122,990 ± 0.009% |
| Dijkstra1000 | 125 ± 2% | 343.9 ± 0.7% | | 51,670 ± 0.1% | 1,972,700 ± 0.04% |
| Dijkstra10000 | 412.8 ± 0.6% | 21,211 ± 0.03% | | 215,940 ± 0.09% | X |
|  | | Linear | Tree | | |
| Database500x2500 | 123 ± 4% | 150.9 ± 0.7% | 1,663.0 ± 0.09% | 18,250 ± 0.2% | 327,390 ± 0.01% |
| Database1000x2500 | 116 ± 3% | 224 ± 1% | 3,218.6 ± 0.09% | 45,020 ± 0.1% | 631,200 ± 0.05% |
| Database1500x5000 | 117 ± 1% | 356.2 ± 0.3% | 8,300.1 ± 0.05% | 112,800 ± 0.2% | X |
| Database5000x5000 | 116.4 ± 0.9% | 1,398 ± 0.1% | 31,037 ± 0.05% | 351,800 ± 0.2% | X |
| Database5000x25000 | 146 ± 1% | 3,538.5 ± 0.03% | 91,919 ± 0.04% | 1,690,000 ± 0.1% | X |

Table 1: Execution times (in ms) for each protocol, benchmarked with the Unix `time` command. All SGX programs were run for 100 iterations, and GC programs were run for 10 iterations. X's represent runs that would not complete in a timely manner. -'s indicate programs we did not run due to hybrid Millionaires not being implemented.

DATABASE. The database holds 64-bit entries. We experiment using both the tree-based ORAM (described in Section 6) and a simple linear search for comparison (marked as *tree* and *linear* in Table 1). For example, `Database500x2500` executes 2,500 select[20] queries on 500 entries. This program demonstrates the time it would take to use a database in a larger application with our protocols. As many programs set and modify entries of the database multiple times in the same run, we have more queries than the size of the database in our tests to explore the efficiency as the query size increases.

For our tests, we establish 5% of the queries to be highly-sensitive and only ever entered into the garbled circuit in even rounds.

DIJKSTRA. The graph contains $n$ nodes, each with 4 edges. Edge weights are 32 bits. For example, `Dijkstra20` considers 20 nodes.

To test this setup, we define the number of (1) nodes in the less sensitive portion of the graph, (2) nodes in the more sensitive portion of the graph, and (3) entrances and exits from the more sensitive portion of the graph. Depending on the start and end points, a route may or may not need to traverse the sensitive portion of the graph. Even if the route does not, we evaluate the circuit to avoid leaking information about the path computed in the enclave.

Test cases are labeled according to the number of nodes in the less sensitive graph. We run the following tests: for the 20-node less sensitive graph, we consider 12-entrances or exits to the sensitive graph and a 20-node sensitive graph. We also run similar tests with the following configurations: 50 (12, 20), 100 (22, 25), 200 (32, 50), 250 (42, 100), 250 (42, 100), 1000 (52, 150), and 10000 (62, 250).

MILLIONAIRES. Inputs are two $n$-bit unsigned integers. Output is a single bit, informing each party whose input is larger. For example, `Millionaires1024` takes 1024 bits of input from each party.

## 8.2 Results

Table 1 presents our results, which we summarize below.

HYBRID SFE-SGX VS. GC-BASED SFE. By only requiring part of the computation to use a garbled circuit, we can increase performance by up to 38x versus pure GC-based SFE in the case of Dijkstra with 1000 nodes. The Database application also demonstrated noticeable improvements, being 18x faster in the 500x2500 case.

Although the hybrid SFE-SGX protocol requires more rounds of computation by virtue of the splitting between even (GC) and odd (SGX) rounds, it achieves less communication overhead than pure GC-based SFE. In odd rounds, the amount of data exchange is much smaller, with there being no oblivious transfer or relaying back of lengthy garbled outputs. The enclave owner may communicate with its enclave at little or no cost, while the other party may communicate with the enclave over the secure channel. This exchange handled in the odd rounds lets us reduce the size of the garbled circuit transmitted in even rounds.

The increase in performance is also dependent upon the size of the computation, determined by each user's requirements for their data. We expect the improvement would be even more substantial for `Dijkstra10000` and larger database cases which were not run.

HYBRID SFE-SGX VS. SGX-ENABLED SFE. Our hybrid SFE-SGX protocol combines garbled circuits with SGX-enabled SFE, which includes side-channel protections. All else being the same, use of a garbled circuit results in a performance reduction vs. SGX-enabled SFE by up to 150x in the worst case, Dijkstra with 1000 nodes.

SGX-ENABLED SFE VS. GC-BASED SFE. Our SGX-enabled SFE protocol is up to 5736x faster than a pure GC execution of `Dijkstra1000`. In the worst case, the SGX-enabled SFE program is only 6x faster; this is for the smallest experiment, a Millionaires program with 1024 bits of input. In larger programs, our results show drastic improvements compared to the garbled circuit implementation.

---

that SFE could be achieved for functions requiring more than two rounds. We keep our evaluation versions of these applications at two rounds, as this is sufficient to show the overhead of transitioning between SGX and GC evaluation.

[20]The time-complexity for set and select queries is the same in this setup, though it requires the addition of an extra 64 bits of input per set.

SGX-enabled SFE vs Naïve SGX-enabled SFE. Our results show, unsurprisingly, that without memory protections, the naïve SGX-enabled SFE protocol outperforms the SGX-enabled SFE protocol for both Dijkstra and Database, due to the necessity of hiding the memory access pattern in SGX-enabled SFE, by anywhere from 1.04x (for `Dijkstra50`) to 630x (for `Database5000x25000` with Tree ORAM). The runtime for Millionaires is almost the same for both protocols.

ORAM vs Linear Search. Our results show that the overhead of the tree ORAM we created is not competitive with a simple linear search for the program sizes we were able to test; this is somewhat surprising given the difference in the fraction of the database that must be searched. The speed of the tree-based ORAM was reduced (from the typical $O(polylog(n))$ complexity for ORAM schemes) for several reasons, including increased time to delete each node from the tree, and time to mix the node data after it has been removed from the tree ($O(n^2)$-time complexity). Thus the linear search is faster for an array database due to a number of factors including the overhead of the ORAM, branch prediction, and caching.

## 9 RELATED WORK

**SFE** Computation on secure data has long been a goal of the theory and systems communities. Mechanisms such as Yao's garbled circuit protocol [69] provided proof that arbitrary secure computation was possible, but proved too inefficient for practical use. More than two decades later, Fairplay [42] provided the first practically efficient implementation of this construction. Since then, a variety of GC-based SFE protocols have been developed in the semi-honest [26, 38, 39], covert [4, 22, 44] and malicious [37, 40, 56] adversarial models. Combined with other efforts to reduce bandwidth [13, 33] and circuit size [8, 36], execution times of applications using garbled circuits for secure computation have dropped by over five orders of magnitude in the last decade. A range of privacy-preserving versions of applications have thus been created, including databases [21], navigation [13, 67], biometrics [11, 14], and genomics [26]. However, these applications still introduce substantial computational overhead. Bahmani et. al [5] share our goal of using SGX to achieve secure computation but leave the entire computation to the enclave. Ohrimenko et. al [46] similarly rely entirely on SGX for multi-party machine learning. We realize the limitations of such a naïve usage of SGX and instead propose partition-relative SFE that combines SGX with garbled circuits for stronger assurances.

**SGX** Most SGX solutions are in favor of putting the whole application or libOS into an enclave, including Haven [6], SCONE [2], Graphene-SGX [62], and Ryoan [27], putting full trust into SGX and holding a large TCB inside the enclave. While Intel SGX SDK and Panoply [58] mandate program partitioning to minimize the TCB, the assumption is the security guarantee of enclaves would not be compromised. A number of attacks have been demonstrated against SGX. AsyncShock [65] exploits synchronization bugs in enclaves using a pre-release version of the Intel SGX SDK. Controlled-channel attacks [68] use memory access patterns to exfiltrate sensitive information from secure enclaves. Cache-based side-channel attacks [10, 53] are also shown possible. Other side-channel vulnerabilities [66] are also found within the Integrated Performance Primitives (IPP) cryptographic library used by Intel SGX SDK. Micro-architecture attacks have also proven to work on enclaves; these include Meltdown [41] and Spectre [34]. Foreshadow [63] attacks extract the attestation key from enclaves thus breaking SGX remote attestation. All these attacks show that SGX can be compromised. Countermeasures include T-SGX [30], SGX-Shield [54], ROTE [43], Sanctum [19], etc., ranging from software enhancement to hardware changes. While these defenses are useful to secure SGX applications, our hybrid approach allows SGX to be compromised by leveraging SFE to secure the most sensitive computation.

**Others** Employing the SGX enclave to securely perform some or all of the evaluation of a function $f$ echoes secure outsourcing. Chaum and Pedersen first proposed the idea of outsourcing using "wallets with observers" [15]. Hohenberger and Lysyanskaya introduced the untrusted but much more computationally powerful worker [25]. Follow-on works founded the idea of a verifiable, outsourceable computation scheme [23] and further improved on this idea [20, 47, 55]. Tramèr et al. [61] previously modeled the enclave as a black-box, albeit in a weaker sense, by keeping only critical functionalities secret from the host. Town Crier [71] also treats the enclave as a black-box, trusted for confidentiality and integrity. From a theoretical standpoint, our notion of partition-relative SFE is closely related to the study of non-simultaneous SFE initiated by Halevi, Lindell, and Pinkas [24]. Their result justifies our assertion that the notion of partition-relative SFE is the strongest-possible security objective in our setting.

## 10 CONCLUSION

Secure function evaluation (SFE) allows mutually distrusting parties to compute the result of a function without leaking anything to others but suffers from large runtime overhead. Intel SGX provides a natural environment for secure computation, but side-channel and micro-architecture attacks can leak data from enclaves. We propose a hybrid approach to combine SFE and SGX, by formalizing what it means to partition the computation of a function $f$ into a piece evaluated with SGX and another piece evaluated using SFE (garbled circuits). We defined secure evaluation of a function relative to such a partitioning of that function and proved our scheme achieves this notion. Our comprehensive evaluation of two case studies shows that we achieve a 38x speedup over traditional garbled circuit methods with our SGX-enabled SFE and points to a way to ensure safer and faster secure computation.

## REFERENCES

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of HASP*, 2013.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, and Others. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of OSDI*, 2016.

[3] A. Atamli-Reineh and A. Martin. Securing Application with Software Partitioning: A Case Study Using SGX. In *Proceedings of SecureComm*, 2015.

[4] Y. Aumann. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *J. Cryptology*, 18(3):554–343, 2010.

[5] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. Secure Multiparty Computation from SGX. Cryptology ePrint Archive, Report 2016/1057, 2016.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[7] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. In *Proceedings of ACM STOC*, 1990.

[8] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *Proceedings of IEEE S&P*, 2013.

[9] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of Garbled Circuits. In *Proceedings of ACM CCS*, 2012.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of USENIX WOOT*, 2017.

[11] J. Bringer, H. Chabanne, M. Favre, A. Patey, T. Schneider, and M. Zohner. GSHADE: Faster Privacy-Preserving Distance Computation and Biometric Identification. In *Proceedings of ACM IH&MMSec*, 2014.

[12] B. B. Brumley and R. M. Hakala. Cache-Timing Template Attacks. In *Proceedings of ASIACRYPT*, 2009.

[13] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of USENIX Security*, 2013.

[14] H. Carter, B. Mood, P. Traynor, and K. Butler. Outsourcing Secure Two-Party Computation as a Black Box. In *Proceedings of CANS*, 2015.

[15] D. Chaum and T. P. Pedersen. Wallet Databases with Observers. In *Proceedings of CRYPTO*, 1993.

[16] J. I. Choi, D. J. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor. A Hybrid Approach to Secure Function Evaluation using SGX. *arXiv Computing Research Repository (CoRR)*, abs/1905.01233, 2019.

[17] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure Web Applications via Automatic Partitioning. In *Proceedings of ACM SOSP*, 2007.

[18] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):23:1–23:20, Jan. 2012.

[19] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of USENIX Security*, 2016.

[20] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, et al. Geppetto: Versatile Verifiable Computation. In *Proceedings of IEEE S&P*, 2015.

[21] 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 WAHC*, 2014.

[22] I. Damgård, M. Geisler, and J. B. Nielsen. From Passive to Covert Security at Low Cost. In *Proceedings of TCC*, 2010.

[23] R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of CRYPTO*, 2010.

[24] S. Halevi, Y. Lindell, and B. Pinkas. Secure Computation on the Web: Computing without Simultaneous Interaction. In *Proceedings of CRYPTO*, 2011.

[25] S. Hohenberger and A. Lysyanskaya. How to Securely Outsource Cryptographic Computations. In *Proceedings of TCC*, 2005.

[26] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of USENIX Security*, 2011.

[27] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of OSDI*, 2016.

[28] Intel Corporation. Intel® Software Guard Extensions Enclave Writer's Guide, 2015. Revision 1.02.

[29] Intel Corporation. Intel Software Guard Extensions for Linux OS. https://01.org/intel-softwareguard-eXtensions, 2017. [Online].

[30] S. Jaebaek, L. Byoungyoung, K. Sungmin, S. Ming-Wei, S. Insik, H. Dongsu, and K. Taesoo. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of NDSS*, 2017.

[31] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of NDSS*, 2016.

[32] S. Johnson. Intel® SGX and Side-Channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, 2017. [Online].

[33] S. Kamara, P. Mohassel, and B. Riva. Salus: A System for Server-Aided Secure Function Evaluation. In *Proceedings of ACM CCS*, 2012.

[34] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, et al. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of IEEE S&P*, 2019.

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

[36] B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of USENIX Security*, 2013.

[37] B. Kreuter, a. shelat, and C.-H. Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of USENIX Security*, 2012.

[38] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *Proceedings of ACM CCS*, 2006.

[39] Y. Lindell and B. Pinkas. Privacy Preserving Data Mining. In *Proceedings of CRYPTO*, 2000.

[40] Y. Lindell and B. Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In *Proceedings of TCC*, 2011.

[41] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, et al. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of USENIX Security*, 2018.

[42] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–A Secure Two-Party Computation System. In *Proceedings of USENIX Security*, 2004.

[43] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of USENIX Security*, 2017.

[44] A. Miyaji and M. S. Rahman. Privacy-Preserving Data Mining in Presence of Covert Adversaries. In *Proceedings of ADMA*, 2010.

[45] 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 Euro S&P*, 2016.

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

[47] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of IEEE S&P*, 2013.

[48] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation is Practical. In *Proceedings of ASIACRYPT*, 2009.

[49] M. O. Rabin. How To Exchange Secrets with Oblivious Transfer. Cryptology ePrint Archive, Report 2005/187, 2005.

[50] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of USENIX Security*, 2015.

[51] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 571–582. ACM, 2013.

[52] T. Schneider and M. Zohner. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *Proceedings of FC*, 2013.

[53] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of DIMVA*, 2017.

[54] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of NDSS*, 2017.

[55] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the Conflict Between Generality and Plausibility in Verified Computation. In *Proceedings of EuroSys*, 2013.

[56] a. shelat and C.-H. Shen. Fast Two-Party Secure Computation with Minimal Assumptions. In *Proceedings of ACM CCS*, 2013.

[57] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of NDSS*, 2017.

[58] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of NDSS*, 2017.

[59] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of ACM CCS*, 2015.

[60] 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 ACM CCS*, 2013.

[61] F. Tramèr, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *Proceedings of Euro S&P*, 2017.

[62] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of USENIX ATC*, 2017.

[63] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of USENIX Security*, 2018.

[64] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of ACM CCS*, 2017.

[65] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of ESORICS*, 2016.

[66] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Proceedings of ACSAC*, 2018.

[67] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell. Privacy-Preserving Shortest Path Computation. In *Proceedings of NDSS*, 2016.

[68] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of IEEE S&P*, 2015.

[69] A. C. Yao. Protocols for secure computations. In *Proceedings of IEEE FOCS*, 1982.

[70] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure Program Partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.

[71] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of ACM CCS*, 2016.

# A APPENDIX

Let $[1..\ell]$ denote the set of integers from 1 to $\ell$. If $\mathbf{a}$ is a vector over strings, let $\text{len}(\mathbf{a})[i] = |\mathbf{a}[i]|$ for every $i \in [1..|\mathbf{a}|]$. A function $\epsilon(\cdot)$ is *negligible* if for every positive polynomial $p(\cdot)$, there exists a $k_0$ such that for every $k \geq k_0$, it holds that $\epsilon(k) < 1/p(k)$. Both an *adversary* and a *simulator* are randomized algorithms.

## A.1 Protocols

We define syntax and execution semantics for two-party protocols in which one or both players have access to an explicitly defined oracle. An *oracle-relative, two-party protocol* is a triple $\Pi = (\text{Init}, \text{Proc}, \mathcal{O})$ where Init and Proc are randomized algorithms and $\mathcal{O}$ is an oracle. Let 0 and 1 denote the (non-oracle) protocol parties.

Fix security parameter $k \geq 0$. Algorithm Init takes as input $1^k$ and outputs a triple of strings $(L_0, L_1, L_{\mathcal{O}})$ called the *long-term inputs* of parties 0, 1 and oracle $\mathcal{O}$ respectively. This is written $(L_0, L_1, L_{\mathcal{O}}) \leftarrow^{\$} \text{Init}(1^k)$.

Algorithm Proc takes as input the identity $i$ of a message recipient, their long-term input $L_i$, their current state $st_i$, and the message $m$ being delivered. It outputs a message $m'$ (to be sent to $1-i$), the coins $w_i$ used in the computation, the updated state $st'_i$, and an indication of whether to halt ($\bot$) or continue ($\top$). The algorithm is given oracle access to $\mathcal{O}(L_{\mathcal{O}}, \cdot)$. On input $(L_{\mathcal{O}}, x)$, the oracle performs some operation (specified by the protocol) on the string $x$ and its current state, updates its state, and returns a string to the player. This is written $(m', w_i, st'_i, \delta) \leftarrow^{\$} \text{Proc}_i^{\mathcal{O}}(L_i, st_i, m)$ where $\delta \in \{\bot, \top\}$.

Protocols are executed with respect to the players' private inputs. Executing the protocol with input $(a_0, a_1)$, where $a_0$ and $a_1$ are strings (or vectors over strings) means to initialize $st_i = (1^k, a_i)$ for $i \in \{0, 1\}$, and facilitate the exchange of messages between 0 and 1 until both halt. More formally, the protocol consists of a sequence of calls to $\text{Proc}_i^{\mathcal{O}}(\cdot, \cdot, \cdot)$ alternating between $i = 0$ and $i = 1$. The protocol specifies which player starts. This is denoted

$$(y_0, y_1, \pi, st') \leftarrow^{\$} \Pi(1^k, a, b, st).$$

The transcript consists of the inputs $1^k$, $a_0$, $a_1$, and $st$, the outputs of each call to Proc, and all of the oracle queries made by Proc, along with their responses.

Let $\omega = \text{View}_{\Pi, k}^i(\pi)$ denote the view of player $i$ for the particular execution of $\Pi$ transcribed by $\pi$. In particular, string $\omega$ encodes $i$'s initial state, and the portions of $\pi$ that correspond to an execution $\text{Proc}_i^{\mathcal{O}}(\cdot, \cdot, \cdot)$, i.e. those corresponding to party $i$.

COMPOSING ORACLE PROTOCOLS. Let $(\Pi_1, \ldots, \Pi_\ell)$ be a sequence of protocols. A protocol $\Pi$ constructed from their composition is defined as follows. The initialization algorithm $\Pi.\text{Init}$ executes each $\Pi_j.\text{Init}$ and returns the triple $(\mathbf{L}_0, \mathbf{L}_1, \mathbf{L}_{\mathcal{O}})$ where $\mathbf{L}_0[j]$ (resp. $\mathbf{L}_1[j]$ and $\mathbf{L}_{\mathcal{O}}[j]$) denotes the long-term input of player 0 (resp. player 1 and oracle $\Pi_j.\mathcal{O}$) in protocol $\Pi_j$. Executing $\Pi$ on inputs $(1^k, a_0, a_1, st)$, where $a_0$ and $a_1$ are strings (or vectors over strings) means to iteratively execute $\Pi_j$ for each $j$ from 1 to $\ell$ on inputs specified by the protocol, except that $st$ is the initial state of oracle $\Pi_1.\mathcal{O}$. During the execution of $\Pi_j$, the algorithm $\Pi_j.\text{Proc}$ is given oracle access to $\Pi_j.\mathcal{O}(\mathbf{L}_{\mathcal{O}}[j], \cdot)$. The output is the tuple $(y_0, y_1, \pi, st')$ where

$y_0$ and $y_1$ are the final states of players 0 and 1 respectively after executing $\Pi_\ell$, string $\pi = (\pi_1, \ldots, \pi_\ell)$, where $\pi_j$ is the transcript of the $j$-th protocol, and $st'$ is the final state of oracle $\Pi_\ell.\mathcal{O}$. Finally, let $\text{View}_{\Pi, k}^i(\pi) = (\text{View}_{\Pi_j, k}^i(\pi_j))_{j=1}^\ell$.

## A.2 Garbling schemes

A garbling scheme $\mathcal{G} = (\text{Gb}, \text{En}, \text{Ev}, \text{De})$ is *projective* if $e$ (the second output of the garbling algorithm) is a sequence $(X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$ of strings called *tokens* such that $\text{En}(e, x) = (X_1^{x_1}, \ldots, X_n^{x_n})$ where $x = x_1 \cdots x_n$.

Let $\mathcal{G}$ be a projective garbling scheme, $f : \{0, 1\}^n \to \{0, 1\}^*$ be a function, and $(F, e, d)$ be a triple of strings in the range of $\text{Gb}(1^k, f)$. Let $\ell \leq n$ be an integer, and let $r = n - \ell$. Then if $a \in \{0, 1\}^\ell, b \in \{0, 1\}^r$, and $x = a \| b$, we define $\text{EnA}(e, a)$ and $\text{EnB}(e, b)$ so that $\text{En}(e, x) = \text{EnA}(e, a) \| \text{EnB}(e, b)$. That is, $\text{EnA}(e, a) = (X_1^{a_1}, \ldots, X_\ell^{a_\ell})$ and $\text{EnB}(e, b) = (X_{\ell+1}^{b_1}, \ldots, X_n^{b_r})$.

PRIV. We specify the simulation-based notion of [9] (instantiated in our setting) in the bottom-left panel of Figure 9. This game captures an adversary's advantage in distinguishing the output of the garbling algorithm on input $(f, x)$ from the output of the simulator on input $(f, f(x))$. A garbling scheme is "secure" if for every reasonable adversary, there exists a simulator such that this advantage is "small". Intuitively, this captures the idea that if a garbling scheme is secure, then possession of the garbled function, garbled input, and the final output leaks only a negligible amount information about the input $x$ to the circuit evaluator. We define the advantage of $\mathcal{A}$ in the game with simulator $\mathcal{S}$ at security parameter $k$ as

$$\text{Adv}_{\mathcal{G}}^{\text{priv}}(\mathcal{A}, \mathcal{S}, k) = 2 \cdot \Pr\left[\text{Exp}_{\mathcal{G}}^{\text{priv}}(\mathcal{A}, \mathcal{S}, k) = 1\right] - 1.$$

We say that $\mathcal{G}$ is *private* if for every polynomial-time adversary $\mathcal{A}$, there exists a polynomial-time simulator $\mathcal{S}$ such that the function $\text{Adv}_{\mathcal{G}}^{\text{priv}}(\mathcal{A}, \mathcal{S}, k)$ is a negligible function of $k$.

## A.3 1-2 oblivious transfer

A *1-2 transfer protocol* is a two-party protocol $\Pi$ played by Alice and Bob with the following correctness condition: when executed with Bob's private input $b = b_1 \cdots b_n \in \{0, 1\}^n$ and Alice's private input of a sequence of tokens $(X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$, it holds that

$$\Pr\left[\text{Out}_{\Pi, k}^{\text{Bob}}((X_1^0, X_1^1, \ldots, X_n^0, X_n^1), b) = (X_1^{b_1}, \ldots, X_n^{b_n})\right] = 1$$

and

$$\Pr\left[\text{Out}_{\Pi, k}^{\text{Alice}}((X_1^0, X_1^1, \ldots, X_n^0, X_n^1), b) = \varepsilon\right] = 1,$$

where $|X_j^0| = |X_j^1|$ for each $j \in [1..n]$.

We define oblivious transfer (OT) in the presence of a semi-honest adversary. Following [9, section 7.1], we formulate the security of OT as an instance of *private function evaluation*, or PFE. Here Alice has private function $f$ and Bob has a private input $x$ in the domain of $f$. The goal is that Bob learns $f(x)$ without learning anything about $f$ (except $|f|$) and Alice learns noting about $x$ (except $|x|$). In the case of 1-2 OT, the private function corresponds to the map $b \mapsto (X_1^{b_1}, \ldots, X_n^{b_n})$ (and hence encodes Alice's tokens) and the private input is Bob's string $b$. The length of this map is a function of $|(X_1^0, X_1^1, \ldots, X_n^0, X_n^1)|$.

| | |
|---|---|
| $\text{Exp}^{\text{sfe}}_{\Pi,\mathcal{P},f,i}(\mathcal{A},\mathcal{S},k)$ | $\text{Exp}^{\text{sfe-odd}}_{\Pi,f,i}(\mathcal{A},\mathcal{S},k)$ |
| $c \leftarrow\!\!\$\ \{0,1\}$ <br> $(a,b,\sigma) \leftarrow\!\!\$\ \mathcal{A}(\text{pick}, 1^k, f)$ <br> if $(a,b) \notin \text{dom} f$ then return $\perp$ <br> $(L_0, L_1, L_{\mathcal{O}}) \leftarrow\!\!\$\ \Pi.\text{Init}(1^k); (\mathbf{a}, \mathbf{b}, \mathbf{y}_0, \mathbf{y}_1) \leftarrow\!\!\$\ \text{Exec}_{\mathcal{P},k}(a,b)$ <br> $(z_0, z_1, \pi, st') \leftarrow\!\!\$\ \Pi(1^k, \mathbf{a}, \mathbf{b}, \varepsilon)$ <br> if $c = 1$ then $\omega \leftarrow \text{View}^i_{\Pi,k}(\pi)$ <br> else if $i = \text{Alice}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, \mathbf{a}, \text{len}(\mathbf{b}), \mathbf{y}_0)$ <br> else if $i = \text{Bob}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, \text{len}(\mathbf{a}), \mathbf{b}, \mathbf{y}_1)$ <br> $c' \leftarrow\!\!\$\ \mathcal{A}(\text{guess}, \sigma, \omega)$ <br> return $(c = c')$ | $c \leftarrow\!\!\$\ \{0,1\}$ <br> $(a,b,st,\sigma) \leftarrow\!\!\$\ \mathcal{A}(\text{pick}, 1^k, f)$ <br> if $(a,b,st) \notin \text{dom} f$ then return $\perp$ <br> $(L_0, L_1, L_{\mathcal{O}}) \leftarrow\!\!\$\ \Pi.\text{Init}(1^k)$ <br> $(y_0, y_1, \pi, st') \leftarrow\!\!\$\ \Pi(1^k, a, b, st)$ <br> if $c = 1$ then $\omega \leftarrow \text{View}^i_{\Pi,k}(\pi)$ <br> else if $i = \text{Alice}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, a, |b|, y_0, |st|)$ <br> else if $i = \text{Bob}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, |a|, b, y_1, |st|)$ <br> $c' \leftarrow\!\!\$\ \mathcal{A}(\text{guess}, \sigma, \omega)$ <br> return $(c = c')$ |
| $\text{Exp}^{\text{priv}}_{\mathcal{G}}(\mathcal{A},\mathcal{S},k)$ | $\text{Exp}^{\text{pfe}}_{\Pi,i}(\mathcal{A},\mathcal{S},k)$ |
| $c \leftarrow\!\!\$\ \{0,1\}$ <br> $(f,x,\sigma) \leftarrow\!\!\$\ \mathcal{A}(\text{pick}, 1^k)$ <br> if $x \notin \text{dom} f$ then return $\perp$ <br> if $c = 1$ then $(F,e,d) \leftarrow\!\!\$\ \text{Gb}(1^k, f); X \leftarrow \text{En}(e,x)$ <br> else $(F,X,d) \leftarrow\!\!\$\ \mathcal{S}(1^k, f, f(x))$ <br> $c' \leftarrow\!\!\$\ \mathcal{A}(\text{guess}, F, X, d, \sigma)$ <br> return $(c = c')$ | $c \leftarrow\!\!\$\ \{0,1\}$ <br> $(f,x,\sigma) \leftarrow\!\!\$\ \mathcal{A}(\text{pick}, 1^k)$ <br> if $x \notin \text{dom} f$ then return $\perp$ <br> $(L_0, L_1, L_{\mathcal{O}}) \leftarrow\!\!\$\ \Pi.\text{Init}(1^k)$ <br> $(y, \varepsilon, \pi, st') \leftarrow\!\!\$\ \Pi(1^k, f, x, \varepsilon)$ <br> if $c = 1$ then $\omega \leftarrow \text{View}^i_{\Pi,k}(\pi)$ <br> else if $i = \text{Alice}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, f, |x|, \varepsilon)$ <br> else if $i = \text{Bob}$ then $\omega \leftarrow\!\!\$\ \mathcal{S}(1^k, |f|, x, f(x))$ <br> $c' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}}(\text{guess}, \sigma, \omega)$ <br> return $(c = c')$ |

**Figure 9: Security notions for function evaluation (top-left), odd-round protocols (top-right), private garbling schemes (bottom-left), and private function evaluation (bottom-right). Let $\Pi = (\text{Init}, \text{Proc}, \mathcal{O})$ be an oracle protocol and $\mathcal{G} = (\text{Gb}, \text{En}, \text{Ev}, \text{De})$ be a garbling scheme. See the Appendix for additional notation.**

**PFE.** Security is captured by an experiment defined in the bottom-right of Figure 9 associated to player $i \in \{\text{Alice}, \text{Bob}\}$, adversary $\mathcal{A}$, and simulator $\mathcal{S}$. If $i = \text{Alice}$, the goal of the adversary is to distinguish Alice's view from the output of $\mathcal{S}$, which is given as input the security parameter, the function, and the length of Bob's private input. If $i = \text{Bob}$, the goal of the adversary is to distinguish player Bob's view from the output of $\mathcal{S}$, which is given as input the security parameter, the input $x$, the value $f(x)$, and the length of the private function. The advantage of $\mathcal{A}$ in the game instantiated with simulator $\mathcal{S}$ at security parameter $k$ is defined as

$$\text{Adv}^{\text{pfe}}_{\Pi,i}(\mathcal{A},\mathcal{S},k) = 2 \cdot \text{Pr}\left[\text{Exp}^{\text{pfe}}_{\Pi,i}(\mathcal{A},\mathcal{S},k) = 1\right] - 1.$$

We say that $\Pi$ is PFE-secure if for each $i \in \{\text{Alice}, \text{Bob}\}$ and every polynomial-time adversary $\mathcal{A}$, there exists a polynomial-time simulator $\mathcal{S}$ such that the function $\text{Adv}^{\text{pfe}}_{\Pi,i}(\mathcal{A},\mathcal{S},k)$ is a negligible function of $k$.

**OT.** We say that a 1-2 transfer protocol is *oblivious* if it is a secure PFE protocol.

## A.4 Symmetric encryption

We give the standard concrete security notion for symmetric encryption. A symmetric encryption scheme $\Gamma$ is a triple of randomized algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D})$. Algorithm $\mathcal{K}$ outputs a string $K$ called the key. Algorithm $\mathcal{E}$ takes as input the key $K$, a message $M \in \{0,1\}^*$, and outputs a ciphertext $C \in \{0,1\}^*$. Algorithm $\mathcal{D}$ takes as input the key $K$, a ciphertext $C$, and deterministically

outputs the message $M$. Correctness demands that for key $K$ and every message $M$ (in the implicit message space), it holds that $\text{Pr}[C \leftarrow\!\!\$\ \mathcal{E}_K(M) : \mathcal{D}_K(C) = M] = 1$.

**IND$.** We associate to an adversary and encryption scheme an experiment in which the adversary is given an oracle, which it may query any number of times. After interacting with its oracle, it outputs a bit. We define the advantage of an adversary $\mathcal{A}$ in attacking $\Gamma$ as

$$\text{Adv}^{\text{ind\$}}_{\Gamma}(\mathcal{A}) = \text{Pr}\left[K \leftarrow\!\!\$\ \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot)} = 1\right] - \text{Pr}\left[\mathcal{A}^{\$(\cdot)} = 1\right]$$

where oracle $\$(\cdot)$ outputs a random bit string of appropriate length, i.e. as long as a ciphertext corresponding to the query would be.