A Formal Foundation for Secure Remote Execution of Enclaves

Pramod Subramanyan
University of California, Berkeley
pramod@berkeley.edu

Rohit Sinha
University of California, Berkeley
rsinha@eecs.berkeley.edu

Ilia Lebedev
Massachusetts Institute of Technology
ilebedev@mit.edu

Srinivas Devadas
Massachusetts Institute of Technology
devadas@mit.edu

Sanjit A. Seshia
University of California, Berkeley
ssesha@eecs.berkeley.edu

ABSTRACT
Recent proposals for trusted hardware platforms, such as Intel SGX and the MIT Sanctum processor, offer compelling security features but lack formal guarantees. We introduce a verification methodology based on a trusted abstract platform (TAP) that formally models idealized enclaves and a parameterized adversary. We present machine-checked proofs showing that the TAP satisfies the three key security properties needed for secure remote execution: integrity, confidentiality and secure measurement. We then present machine-checked proofs showing that SGX and Sanctum are refinements of the TAP under certain parameterizations of the adversary, demonstrating that these systems implement secure enclaves for the stated adversary models.

KEYWORDS
Enclave Programs; Secure Computation; Formal Verification; Confidentiality; Integrity; Remote Attestation

1 INTRODUCTION
A typical computing platform contains large software layers (e.g., OS, hypervisor, firmware) in its trusted computing base (TCB), where numerous exploits have allowed privileged malware to execute [24, 32, 36, 48, 57]. Recognizing this problem, processor designers and vendors are now developing CPUs with hardware primitives, such as Intel SGX [2, 29, 30, 44] and MIT Sanctum [19], for isolating sensitive code and data within protected memory regions, so-called enclaves, that are inaccessible to all other software running on the machine. In this new paradigm, enclaves are the only trusted (hope-fully small) components of an application, and consequently need to be programmed with greater rigor and stronger defenses.

Despite growing interest, there has only been informal security analysis of these enclave platforms. This lack of formalization has a number of consequences. Developers of enclave programs cannot formally reason about security of their programs; incorrect use of hardware primitives can render enclave programs vulnerable to security breaches. Hardware designers cannot formally state security properties of their architectures and are unable to reason about potential vulnerabilities of enclaves running on their hardware.

Going beyond the difficulty of reasoning about specific programs and platforms, this lack of formalization also makes it difficult to compare and contrast potential improvements to these platforms. For example, a number of proposals have developed software defenses to strengthen the security guarantees of Intel SGX enclaves [51, 60, 61]. However, comparing the security guarantees and adversary models (of say T-SGX [60] and Shinde et al. [61]) is difficult without a unified framework for such reasoning. Furthermore, as we move towards a world in which enclave platforms are widespread, it is conceivable that future data centers will support a number of different enclave platforms. Developers will likely tackle this diversity by relying on tools and libraries that provide a common application programming interface (API) for enclave programs while supporting different target platforms. Reasoning about the security guarantees of such toolchains is also challenging without a unified framework. This paper bridges each of the above gaps by presenting a unified formal framework to specify and verify the security properties of enclave platforms.

We address the formal modeling and verification of enclave platforms in three parts. First, we define the properties required for secure remote execution of enclaves. Next we introduce the trusted abstract platform (TAP), an idealization of enclave platforms along with a parameterized adversary model. We present machine-checked proofs showing that the TAP provides secure remote execution against these adversaries. Finally, we present machine-checked proofs demonstrating that formal models of Intel SGX and MIT Sanctum are refinements [12] of the TAP for different parameterizations of the adversary, and thus also provide secure remote execution.

Secure Remote Execution of Enclaves: This paper first formalizes a model of computation of an enclave program, and the attacker’s operations and observations – we assume a privileged software adversary that has compromised the host OS, hypervisor, network, and persistent storage. The execution model allows the developer (or user) to precisely define the expected runtime behavior of an enclave in the presence of a privileged software adversary. When the user outsources an enclave to a remote platform, she seeks a guarantee that the enclave be executed according to the expected behavior, i.e. the platform must respect the enclave’s semantics. We term this property secure remote execution (SRE): any execution of that enclave on the trusted enclave platform must be one of enclave’s expected executions (formalized in § 3). Any enclave platform, such as SGX or Sanctum, must guarantee SRE to the user, and this paper describes a formal framework and methodology to prove secure remote execution. SRE is decomposed into lower-level properties — specifically, integrity, confidentiality, and secure measurement — for which we develop machine-checked proofs on models of SGX and Sanctum.

Trusted Abstract Platform: We develop an idealized abstraction of the aforementioned enclave platforms, named Trusted Abstract Platform (TAP), consisting of a small set of formally-specified primitives sufficient to implement enclave execution. As a precursor to proving that TAP satisfies SRE, we define a parameterized attacker model with varying attacker capabilities. We then present
machine-checked proofs showing that TAP satisfies the integrity, confidentiality, and secure measurement properties for these attackers. The TAP is a general framework for reasoning about and comparing different enclave platforms, adversary models and security guarantees. For enclave platform implementers, the TAP serves as a golden model or specification of platform behavior. From the perspective of enclave program developers, the TAP provides a means of reasoning about program security without being bogged down by implementation details of individual enclave platforms.

**Refinement** Next, we use the TAP to reason about the security of Intel SGX and MIT Sanctum. We develop formal models of SGX and Sanctum and present machine-checked proofs showing that SGX and Sanctum are refinements of our idealized TAP: every operation on SGX and Sanctum can be mapped to a corresponding TAP operation. Since all executions of an enclave on SGX and Sanctum can be simulated by a TAP enclave, and because TAP guarantees SRE, it follows that the SGX and Sanctum models also guarantee SRE. There is a caveat that SGX only refines a version of TAP which leaks some side channel observations to the attacker (see § 5.3), therefore providing a weaker confidentiality guarantee. This form of parameterization demonstrates that the TAP allows us to develop a taxonomy of enclave platforms, each of which provides varying guarantees against different threat models.

### 1.1 Contributions

This paper makes the following key contributions:

1. A formalization of enclave execution in the presence of a privileged software adversary, and secure remote execution (SRE) of an enclave program.
2. The Trusted Abstract Platform, which formally specifies the semantics of a small set of trusted primitives for safely creating and destroying enclaves, entering and exiting from enclaves, and generating attested statements from within enclaves. We also define a parameterized attacker model, for which the TAP gives varying degrees of confidentiality.
3. Decomposition of the SRE property into lower-level properties of integrity, confidentiality, and secure measurement, which we formally specify and verify using a theorem prover.
4. A refinement-based methodology for proving the SRE guarantee of enclave platforms and machine-checked proofs of refinement for models of SGX and Sanctum.

All our models and proof scripts are being made open-source along with this submission [3]. These models are modular and amenable to extension by the community.

### 2 FORMAL MODEL OF ENCLAVE EXECUTION

An enclave platform implements primitives to create protected memory regions, called enclaves, that contain both code and data and are isolated from all other software in the system. The processor monitors all accesses to the enclave: only code running in the enclave can access the enclave’s memory. As an example, Intel’s SGX instructions enable the creation of user-mode enclaves in the hosting application’s address space. Since the privileged software layers (OS-/Hypervisor) cannot be trusted to modify enclave memory (which is why system calls are disabled in enclave mode), the enclave platform allows the enclave to access the entire address space of the hosting application. This enables efficient I/O interaction with the external world. The external world can only transfer control to the enclave at statically-defined locations called entrypoints. In addition to enabling isolated execution, the enclave platform implements primitives for generating attested statements: code inside an enclave can get messages signed using a per-processor private key along with a hash-based measurement of the enclave. This allows other trusted entities to verify that messages originated from the desired enclave running on a genuine platform. Finally, we assume the enclave platform implements a cryptographically secure random number generator which the enclave can use for cryptographic operations such as key generation.

![Figure 1: Execution of Enclave Program.](image)

To outsource the enclave’s execution, the user sends the enclave program to a remote machine over an untrusted channel (Figure 1). The untrusted OS invokes the enclave platform’s primitives to launch an enclave containing the program. While running, an enclave may invoke the enclave platform’s primitives to get attested statements and random bits. The enclave may also send outputs to the user by proxying them via the host application’s unprotected memory. We present a sample application in § 3.3 to discuss details of secure remote execution of an enclave program.

#### 2.1 Formal Model of Enclave Programs

An enclave is launched with a set of code pages, containing user-mode instructions, and a set of data pages, which can be used for private heap and stack space. This forms the initial state of the enclave — the enclave platform includes a hash-based measurement of this initial state in all attested statements, which allows the user to verify that the enclave was launched in an expected initial state. The enclave’s configuration also includes (1) the entrypoint, (2) the virtual address range `evarange` which maps the enclave’s protected memory (i.e., each address in `evarange` either maps to a physical address that is owned by the enclave or is inaccessible) and (3) permissions for each address within `evarange`.

**Enclave Programs:** The user ships an enclave $e = (init_e, config_e)$. `config_e` defines the enclave’s entrypoint: `config_e.entrypoint`, its virtual address range: `config_e.evarange`, and its access permissions:
which is copied verbatim from an attacker runs concurrently with the enclave, we assume reads from non-enclave memory for reading in-the current state of the enclave platform first identifies the instruction to execute (based on the current state of vmem and pc), and then transitions to the next state based on the operational semantics of the platform’s instructions. The instructions include bitvector operations on registers, memory accesses, and the enclave primitives for generating attested statements, randomness, and exiting enclave mode — the platform also includes privileged instructions (e.g., MSR instructions in x86) which can cause arbitrary updates to $A_e(\sigma)$. This model of execution lets us define the platform’s transition relation $\sim$, where $(\sigma_i, \sigma_j) \in \sim$ indicates that the platform can transition from $\sigma_i$ to $\sigma_j$; from hereon, we write this in infix form as $\sigma_i \sim \sigma_j$. We treat $\sim$ as a relation as opposed to a function to allow non-determinism for the platform. The non-determinism includes randomness from entropy sources, direct memory accesses from I/O peripherals, etc. Let us refer to these bits of non-determinism in a particular state as $I^P(\sigma)$, which is only available to the privileged software — the enclave’s source of non-determinism is captured in $L_e(\sigma)$. We require that the platform be deterministic relative to $I^P(\sigma)$. A secure platform must also ensure an enclave program $e$ is deterministic relative to its input $I_e(\sigma)$. We state two properties in § 3 that imply these determinism guarantees.

### 2.2 Formal Model of the Adversary

An enclave executes in the presence of a privileged adversary that has compromised all software layers including the OS, except for the enclave platform. In this section, we abstractly define the effect of the adversary’s operations and observations. § 4.2 precisely defines the set of attacker operations and observations for the TAP.

#### 2.2.1 Adversary Tampering

The privileged adversary may pause the enclave at any time, and execute arbitrary instructions that modify the attacker’s state $A_e(\sigma)$, enclave’s input $I_e(\sigma)$ for any enclave $e$, and launch or destroy any number of enclaves. We model the adversary’s effect through the “tamp” relation over pairs of states: $(\sigma_1, \sigma_2) \in \text{tamp}$ if the attacker can change the machine’s state from $\sigma_1$ to $\sigma_2$, with the constraint that $E_e(\sigma_1) = E_e(\sigma_2)$. That is, the attacker may modify $A_e(\sigma_1), I_e(\sigma_1),$ and $O_e(\sigma_1)$, but not the enclave’s state $E_e(\sigma_1)$. The constraint that tamper does not affect an enclave’s state is assumed while defining the enclave’s semantics. Our integrity property (§ 3) for enclave platforms must prove that this constraint is forced upon the attacker’s operations.

$tamper \subset \sim$ because a software attacker uses the platform’s instructions to update the platform’s state. Furthermore, tamper is reflexive because the adversary can always leave state unmodified: $\varnothing, (\sigma, \sigma) \in \text{tamp}$. In addition to running concurrently with an enclave $e$, the adversary may tamper the machine’s state prior to launch and modify $e$’s launch state init, and configuration configure.

#### 2.2.2 Adversary Observations

Untrusted software may also observe an enclave’s execution, depending on the confidentiality guarantees provided by the enclave platform. At the very least, the adversary observes any output, but it may also observe certain side channels such as memory access patterns. Observations are performed by executing arbitrary instructions (e.g., any x86 instruction) and invoking the platform’s primitives (e.g., launching other enclaves), and then observing the results of these operations. Let $obs_e(\sigma)$ denote the result of an observation for the machine state $\sigma$. For instance, an attacker that only observes outputs enjoys the observation function $obs_e(\sigma) \equiv (O_e(\sigma))$. We specify the observation functions in detail in § 4.2.

### 2.3 Enclave Execution with an Attacker

An execution trace of the platform is an unbounded-length sequence of states denoted $\pi = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$, such that $\forall i, \sigma_i \sim \sigma_{i+1}$
forcing the platform to execute e at any time, we define e’s execution to be the subsequence of states from π where e is executing. To that end, the function \( \text{curr}(\sigma) \) denotes the current mode of the platform, where \( \text{curr}(\sigma) = e \) iff the platform executes enclave e in state \( \sigma \). Using this function, we can filter out the steps in \( \pi \) where e is not executing. We write the resulting sequence as \( \langle \sigma_0', \sigma_1', \ldots, \sigma_m' \rangle \) where \( \text{init}_e(E_e(\sigma_0')) \land \forall i. \text{curr}(\sigma_i') = e \). This subsequence is the enclave’s execution trace: \( \langle I_e(\sigma_0'), E_e(\sigma_0'), O_e(\sigma_0') \rangle, \ldots, \langle I_e(\sigma_m'), E_e(\sigma_m'), O_e(\sigma_m') \rangle \). Since an execution trace of e only includes the steps where e invokes an instruction, the attacker may perform tamper between any two consecutive steps of e’s execution trace. Therefore, we also have the property that \( \forall i. (\sigma_i', \sigma_{i+1}') \in \text{tamper} \). This has the effect of havocing \( A_e(\sigma) \) and \( l_e^J(\sigma) \) in all these steps, thus supplying the enclave with fresh inputs at each step.

The semantics of an enclave e, denoted \([e]\), is the set of finite or infinite execution traces, containing an execution trace for each input sequence, i.e. for each value of non-enclave memory and randomness at each step of execution.

\[
[e] = \{ \langle I_e(\sigma_0'), E_e(\sigma_0'), O_e(\sigma_0') \rangle, \ldots \mid \text{init}_e(E_e(\sigma_0)) \} 
\]  

We must account for all potential input sequences in \([e]\) because e may receive any value of input at any step. We note that \([e]\) may contain traces of any length, and also contain prefixes of any other trace in \([e]\), i.e. it is prefix-closed. We adopt this definition of \([e]\) because the attacker can pause and destroy the enclave at any time; denial of service is not in scope. Due to the determinism property of enclave programs, a specific sequence of inputs \( \langle I_e(\sigma_0'), E_e(\sigma_0'), \ldots, I_e(\sigma_m') \rangle \) uniquely identifies a trace from \([e]\) and determines the expected execution trace of e under that sequence of inputs.

3 SECURE REMOTE EXECUTION OF ENCLAVES

Imagine a user who wishes to outsource the execution of an enclave program e onto a remote platform. The user desires that the platform respect the semantics \([e]\) by executing trace(s) from \([e]\). However, the privileged software layers on the platform are untrusted, therefore the user’s trust is based on guarantees provided by the hardware platform. We propose the following notion of secure remote execution (SRE) of enclaves:

Definition 3.1. Secure Remote Execution of Enclaves. A remote platform performs secure execution of an enclave program e if any execution trace of e on the platform is contained within \([e]\). Furthermore, the platform must guarantee that a privileged software attacker only observes a projection of the execution trace, as defined by the observation function \( \text{obs} \).

It is important to note that SRE does not force the platform to execute e — the attacker may deny service, and this is easily detectable by the user because the attacker cannot forge attested statements as if they originated from the user’s enclave. Nor are we forcing the platform to execute e a fixed number of times. The attacker has the capability to execute e as many times as it wishes, and a user can easily defend against these attacks by refusing to provision secrets to other copies of the enclave. With that said, SRE requires the platform to execute a trace from \([e]\), and recall that \([e]\) only contains enclave executions that start in the initial state of the enclave (see Equation 1). Furthermore, this definition of \([e]\) assumes secure execution of e in that the attacker only affects e’s execution by affecting the inputs, which are assumed to be untrusted anyway — we later state an integrity property of that validates this assumption of the enclave platform.

3.1 Proof Decomposition of SRE

A rational user will outsource the enclave only to a platform that provides a formal guarantee of SRE. To that end, we describe a method for formally verifying that an enclave platform provides SRE to any enclave program. We provide machine-checked proofs of SRE for the TAP in § 4, and show how this applies to Intel SGX and MIT Sandium in § 5. The key idea is to decompose the SRE property into the following set of properties.

- **Secure Measurement**: The platform must measure the enclave program to allow the user to detect any changes to the program prior to execution, i.e. the user must be able to verify that the platform is running an unmodified e.

- **Integrity**: The enclave program’s execution cannot be affected by a privileged software attacker beyond providing inputs, i.e. the sequence of inputs uniquely determines the enclave’s execution trace, and that trace must be allowed by the enclave’s semantics \([e]\).

- **Confidentiality**: A privileged software attacker cannot distinguish between the executions of two enclaves, besides what is already revealed by \( \text{obs} \).

3.1.1 Secure Measurement. During launch, the platform computes a hash of the enclave’s initial contents (\( \text{init} \)) along with relevant configuration bits (\( \text{config} \)). The hash-based measurement acts as a unique identity for the enclave, which follows directly from the collision resistance assumption of the cryptographic hash function, and therefore finds use in authenticating the enclave. Any deviation from the desired enclave program will be detected when the enclave sends an attested statement to the user — we assume that attested statements are produced using a quoting scheme that is unforgeable under chosen message attacks (UF-CMA); we do not model the cryptography of this scheme, and refer the reader to [54] for a formal treatment of this subject. The secure measurement property states that any two enclaves with the same measurement must also have the same semantics: they must produce equivalent execution traces for equivalent input sequences.

Let \( \mu(e) \) be the measurement of enclave e, computed when launching the enclave. The operation must be such that two enclaves with the same measurement have identical states.

\[
\forall \sigma_1, \sigma_2. \text{init}_e(E_{e_1}(\sigma_1)) \land \text{init}_e(E_{e_2}(\sigma_2)) \Rightarrow \mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2) \tag{2}
\]

Next we need to ensure that the if two enclaves \( e_1 \) and \( e_2 \) have the same state, then they produce equivalent execution traces for equivalent input sequences. This is the determinism property we assumed (while defining \([e]\) in 2.1) of the enclave platform, so we
must prove it here.

\[
\forall \pi_1, \pi_2. \quad (3)
\]
\[
\begin{align*}
\left( E_e(\pi_1[0]) = E_e(\pi_2[0]) \right) \quad & \quad \land \\
\forall i. (curr(\pi_1[i]) = e_1) \iff (curr(\pi_2[i]) = e_2) \quad & \quad \land \\
\forall i. (curr(\pi_1[i]) = e_1) \implies I_e(\pi_1[i]) = I_e(\pi_2[i]) \quad & \quad \implies \\
\forall i. E_e(\pi_1[i]) = E_{e_2}(\pi_2[i]) \land O_e(\pi_1[i]) = O_{e_2}(\pi_2[i])
\end{align*}
\]

Equation 3 states that if: (i) the two traces \(\pi_1\) and \(\pi_2\) start with the same initial state for enclaves \(e_1\) and \(e_2\), (ii) if \(\pi_1\) and \(\pi_2\) enter and exit the enclaves in lockstep (i.e., the two enclaves execute for the same number of steps), (iii) if the input sequence to the two enclaves is the same, then the two enclaves execute identically in both traces: they have the same sequence of state and output values.

### 3.1.2 Integrity

The integrity guarantee ensures that the execution of the enclave in the presence of attacker operations is identical to the execution of the program without the attacker’s operations. In other words, the attacker only impacts an enclave’s execution by controlling the sequence of inputs — all other operations, such as controlling I/O peripherals and executing supervisor-mode instructions, have no effect on the enclave’s execution. Any two traces (of the same enclave program) that start with equivalent enclave states and have the same input sequence will produce the same sequence of enclave states and outputs, even though the attacker’s operations may differ in the two traces.

\[
\forall \pi_1, \pi_2. \quad (4)
\]
\[
\begin{align*}
\left( E_e(\pi_1[0]) = E_e(\pi_2[0]) \right) \quad & \quad \land \\
\forall i. (curr(\pi_1[i]) = e) \iff (curr(\pi_2[i]) = e) \quad & \quad \land \\
\forall i. (curr(\pi_1[i]) = e) \implies I_e(\pi_1[i]) = I_e(\pi_2[i]) \quad & \quad \implies \\
\forall i. E_e(\pi_1[i]) = E_e(\pi_2[i]) \land O_e(\pi_1[i]) = O_e(\pi_2[i])
\end{align*}
\]

Figure 2 shows the two traces from the integrity property. The adversary’s steps are labelled \(A_1\) and \(A_2\) while the enclave’s steps are labelled \(e\). Assumptions are annotated in blue, and proof obligations are shown in red. The enclave’s inputs are assumed to be the same in both traces; this is shown by the \(\approx_e\) symbol. The initial state of the two enclaves is assumed to be the same and this is shown by the blue \(\approx_e\) symbol. The attacker performs different actions in each trace, but the integrity proof must show that the enclave’s state and outputs do not differ: \(\forall i. E_e(\pi_1[i]) = E_e(\pi_2[i]) \land O_e(\pi_1[i]) = O_e(\pi_2[i])\). These proof obligations are denoted by the red \(\approx_e\) and \(\approx_O\) symbols.

#### 3.1.3 Confidentiality

The enclave platform must ensure that the attacker does not observe the enclave’s execution beyond what is allowed by the observation function \(obs\), which must include the \(init\) and \(config\) components of the enclave’s description, outputs to non-enclave memory, exit events from enclave mode to untrusted code, and other side-channels leakage as permitted by the observation function \(obs\). The privileged software attacker may use any combination of machine instructions to perform an attack, and the attacker should observe the same results from executing these instructions for any pair of enclaves that produce the same observation via the function \(obs\). In other words, all enclave traces with the same observations, but possibly different enclave states, must be indistinguishable to the attacker. If the platform contains a vulnerability that allows the attacker to observe different values in the two traces, we say that such a platform does not provide confidentiality.

\[
\forall \pi_1, \pi_2. \quad (5)
\]
\[
\begin{align*}
\left( A_e(\pi_1[0]) = A_e(\pi_2[0]) \right) \quad & \quad \land \\
\forall i. curr(\pi_1[i]) = curr(\pi_2[i]) \land I^P(\pi_1[i]) = I^P(\pi_2[i]) \quad & \quad \land \\
\forall i. curr(\pi_1[i]) = e \implies obs_e(\pi_1[i + 1]) = obs_e(\pi_2[i + 1]) \quad & \quad \implies \\
\forall i. A_e(\pi_1[i]) = A_e(\pi_2[i])
\end{align*}
\]

Figure 3 shows the two traces in the confidentiality property. As in Figure 2, the attacker’s steps are labelled \(A\) while the enclave’s steps are labelled \(e_1\) and \(e_2\). The two traces start off in equivalent states but diverge (at state \(\sigma_j\)) because the two enclaves may perform different computation. The enclave’s adversary-visible observations are assumed to be the same in both traces when the enclave is executing; this is shown by the blue \(\approx_{obs}\). Adversary non-determinism \((I^P(\sigma))\) is assumed to be the same in both traces. The two traces eventually merge (to the same platform state) when the enclave is destroyed. The theorem states that adversary state is identical: \(\forall i. A_e(\pi_1[i]) = A_e(\pi_2[i])\); this is illustrated in the red \(\approx_A\). From this we get that the attacker has not learned any additional information (beyond \(obs\)).

### 3.2 Soundness of SRE Decomposition

**Theorem 3.2.** An enclave platform that satisfies secure measurement, integrity, and confidentiality property for any enclave program also satisfies secure remote execution.

**Proof Sketch:** Suppose that the user sends an arbitrary enclave \(e\) to a remote server for execution, and the platform launches enclave \(e_r\).
some time later — because e is sent over an untrusted channel, e may or may not equal \( e_f \). If the user finds \( \mu(e_f) \neq \mu(e) \), then the platform has no obligations to execute a trace from \( [e] \). Otherwise, if \( \mu(e_f) = \mu(e) \), we have that \( [e_f] = [e] \) thanks to the measurement and integrity properties. So, the two programs have identical runtime behaviors, which is a prerequisite for SRE. Finally, confidentiality implies that the attacker’s observation is restricted to \( obs \).

### 3.3 Application of Secure Remote Execution

The measurement and integrity properties guarantee that the remote platform executes a trace from \( [e] \), while the confidentiality property ensures that the attacker does not learn more information than what the enclave wishes to reveal. Together, these three properties imply SRE.

SRE is useful for building practical trusted applications. Consider the problem of executing a batch job securely in the cloud. The user sends an enclave program, which implements a function on sensitive data, to an enclave platform in the cloud. The protocol includes the following steps:

1. The user sends an encrypted program \( e \) to the cloud provider, which launches the program on an enclave platform.
2. The user and enclave establish an authenticated TLS channel via an ephemeral Diffie-Hellman (D-H) exchange.
   - (a) User sends her public parameter \( g^x \) to the enclave, where \( x \) is a randomly generated fresh value.
   - (b) Enclave sends its public parameter \( g^y \) to the user, in the form of an attested statement, thus guaranteeing that a genuine enclave platform launched the expected enclave \( e \).
   - (c) User and enclave compute a shared master secret \( g^{xy} \), and derive symmetric session keys \( s_{ue} \) and \( s_{eu} \), a key for each direction.
3. The user now sends encrypted input to enclave using this shared secret: \( [in]_{s_{ue}} \).
4. The enclave decrypts its input, performs the computation and returns the encrypted result to the user: \( [out]_{s_{eu}} \).

Consider the following security property: the attacker neither learns secret input \( [in] \) nor the secret output \( [out] \). To that end, the user 1) develops an enclave program that only accepts encrypted inputs and sends encrypted outputs, and 2) specifies an observation function \( obs \) where a privileged software adversary is only allowed to view the enclave’s outputs to non-enclave memory — this is acceptable because \( e \) encrypts its outputs.

The measurement guarantees that the user will only establish a channel with the expected enclave on a genuine enclave platform. Integrity ensures that the platform will execute a trace from \( [e] \), thus respecting \( e \)’s semantics. The platform may choose to not launch \( e \) or prematurely terminate \( e \), but such executions will not generate \( [out]_{s_{eu}} \) and hence can be trivially detected by the user. Integrity also ensures that the platform does not rollback the contents of enclave’s memory while it is alive (i.e., not destroyed) as such attacks will cause the enclave’s execution to proceed differently from \( [e] \), and SRE guarantees \([e]\). SRE does not require the platform to defend against rollback attacks on persistent storage. This protection is not needed for the batch service because the enclave does not update \( [in]_{s_{ue}} \), and any tampering to \( [in]_{s_{ue}} \) will fail the cryptographic integrity checks. Finally, confidentiality ensures that the enclave platform only reveals the \( obs \) function of enclave’s execution to the software attacker, which only includes the encrypted outputs. We now have our end-to-end security property.

Should the enclave require state beyond enclave’s memory to perform the job, it would require integrity, freshness, and confidentiality for non-enclave state, which is not covered by SRE. The enclave can implement cryptographic protections (e.g., Merkle tree) and techniques for state continuity [53] to address this concern.

### 4 THE TRUSTED ABSTRACT PLATFORM

The trusted abstract platform (TAP) consists of a processor with a program counter, general purpose registers, virtual address translation and a set of primitives to support enclave execution. In this section, we first introduce a formal model of the TAP. We present a range of adversary models with varying capabilities. We then present a set of machine-checked proofs showing that the TAP satisfies the properties required for secure remote execution: (i) secure measurement, (ii) integrity and (iii) confidentiality.

#### 4.1 TAP Model

Recall that the TAP is modeled as a finite state transition system: \( TAP = (\Sigma, \sim, \text{init}) \). \( \Sigma \) is the set of states, \( \sim \) is the transition relation and \( \text{init} \in \Sigma \) is the initial state.

##### 4.1.1 TAP State Variables

The states \( \Sigma \) of the TAP are a defined as a valuation of the state variables \( Vars \). These variables are described in Table 1. \( pc, regs, mem \) have their usual meanings. \( \text{adr}_\text{map} \) maps individual virtual addresses to physical addresses and permissions. This is unlike a typical processor which uses a page table to map virtual page numbers to physical page numbers. The TAP is an abstraction and must abstract a diverse set of architectures with different page table structures and page sizes. Therefore, it maps each virtual address to a physical address.

The TAP includes a model of cache which is used to show that the TAP preserves confidentiality in the presence of a software adversary attempting cache attacks. The cache model is of a physically-indexed physically-tagged direct-mapped cache. Note that ensuring confidentiality for a direct-mapped cache also ensures confidentiality for set-associative caches. The TAP cache model leaves the mapping of physical addresses to cache sets and mapping of physical addresses to cache tags uninterpreted. In other words, the TAP formalism only requires that the cache set and cache tag for each memory access be deterministic functions of the physical address. The exact function is specified by implementations (refinements) of TAP, which are models of SGX and Sanctum in this paper.

The variable \( \text{current}_\text{eid} \) tracks the enclave currently being executed. \( \text{owner} \) maps each physical address to the enclave which exclusively “owns” it. If \( \text{owner}[p] = e \), only enclave \( e \) can access (fetch/read/write) this word of memory. We abuse notation and use \( e \) to refer to both the “enclave id,” a unique integer assigned by the platform to an enclave, as well the enclave itself. Attempts to access physical address \( p \) by all other enclaves and privileged software are blocked. \( \text{owner}[p] = \text{OS} \) means that address \( p \) is not allocated to any enclave. \( \text{owner} \) corresponds to the EPCM in SGX and the DRAM bitmap in Sanctum. It is the primary mechanism
When entering enclave state and restored when the enclave exits. When an enclave is interrupted and ensures that enclaves cannot be tricked into resuming execution from invalid state. The virtual to physical mappings of both enclave and privileged software. Each of these operations update cache state, set the access bit in addr, and return whether the operation was a hit or a miss in the cache.

The virtual to physical mappings of both enclave and privileged software are controlled using get_addr_map and set_addr_map. As enclave’s memory access pattern can leak via observation of the access/present bits in addr_map, get_addr_map(e, v) must fail (on a secure TAP) for virtual addresses in the set enc_metadata[e], evrange when called from outside the enclave. However, SGX does permit privileged software to access an enclave’s private page tables.

Table 1: Description of TAP State Variables.

<table>
<thead>
<tr>
<th>State Var.</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>VA</td>
<td>The program counter.</td>
</tr>
<tr>
<td>regs</td>
<td>N \rightarrow W</td>
<td>Architectural registers: map from natural numbers to words.</td>
</tr>
<tr>
<td>mem</td>
<td>PA \rightarrow W</td>
<td>The memory: a map from physical addresses to words.</td>
</tr>
<tr>
<td>addr_map</td>
<td>VA \rightarrow (ACL \times PA)</td>
<td>Map from virtual addresses to permissions and physical addresses for current process.</td>
</tr>
<tr>
<td>cache</td>
<td>Set \rightarrow (B \times Tag)</td>
<td>Direct-mapped cache: map from cache sets to valid bits and cache tags.</td>
</tr>
<tr>
<td>current_eid</td>
<td>(E_{id})</td>
<td>Current enclave. current_eid = OS means that no enclave is being executed.</td>
</tr>
<tr>
<td>owner</td>
<td>PA \rightarrow (E_{id})</td>
<td>Map from physical address to the enclave address is allocated to.</td>
</tr>
<tr>
<td>enc_metadata</td>
<td>(E_{id} \rightarrow E_M)</td>
<td>Map from enclave ids to metadata record type ((E_M)).</td>
</tr>
<tr>
<td>os_metadata</td>
<td>(E_M)</td>
<td>Record that stores a checkpoint of privileged software state.</td>
</tr>
</tbody>
</table>

Table 2: Fields of the enc_metadata record.

<table>
<thead>
<tr>
<th>State var.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entrypoint</td>
<td>Enclave entrypoint.</td>
</tr>
<tr>
<td>addr_map</td>
<td>Virtual to physical mapping/permissions.</td>
</tr>
<tr>
<td>excl_vaddr</td>
<td>Set of private virtual addresses.</td>
</tr>
<tr>
<td>measurement</td>
<td>Enclave measurement.</td>
</tr>
<tr>
<td>pc</td>
<td>Saved PC (in case of interrupt).</td>
</tr>
<tr>
<td>regs</td>
<td>Saved registers (in case of interrupt).</td>
</tr>
<tr>
<td>paused</td>
<td>Flag set only when enclave is interrupted.</td>
</tr>
</tbody>
</table>

We introduce a “setting” in the TAP, called priv_mappings, and this insecure behavior is allowed when priv_mappings = false.

Enclave Metadata: Table 2 lists various fields within the enclave metadata record. It stores the entrypoint to the enclave, its virtual to physical mappings and what set of virtual addresses are private to the enclave. The pc and regs fields are used to checkpoint enclave state when it is interrupted. The paused flag is set to true only when an enclave is interrupted and ensures that enclaves cannot be tricked into resuming execution from invalid state.

Privileged Software Metadata: The os_metadata record contains three fields: pc, regs, and addr_map. The pc and regs fields store a checkpoint of privileged software state. These are initialized when entering enclave state and restored when the enclave exits. The addr_map field is the privileged software’s virtual to physical address mapping and associated permissions.

4.1.2 TAP Operations. Table 3 describes the operations supported by the TAP. fetch, load, store work as usual. The platform guarantees that memory owned by enclave e is not accessible to other enclaves or privileged software. Each of these operations update cache state, set the access bit in addr_map, and return whether the operation was a hit or a miss in the cache.

The virtual to physical mappings of both enclave and privileged software are controlled using get_addr_map and set_addr_map. As enclave’s memory access pattern can leak via observation of the access/present bits in addr_map, get_addr_map(e, v) must fail (on a secure TAP) for virtual addresses in the set enc_metadata[e], evrange when called from outside the enclave. However, SGX does permit privileged software to access an enclave’s private page tables.

\[
\begin{align*}
\text{current}_\text{eid} &= \text{OS} \land e \notin \text{enc}_\text{metadata} \\
\text{executable}(m[t]) \land t \in x_v \\
\forall p. p \in x_p &\implies \text{owner}[p] = \text{OS} \\
\forall v. v \in x_v &\implies (\text{valid}(m[v]) \implies m[v]_{\text{PA}} \in x_p) \\
\forall v_1, v_2. (v_1 \in x_v \land v_2 \in x_v) &\implies (m[v_1]_{\text{PA}} \neq m[v_2]_{\text{PA}}) \\
&\implies (\text{launch}_{\text{status}} = \text{success})
\end{align*}
\]

Note: \(m[v]_{\text{PA}}\) refers to physical address that virtual address \(v\) points to under the mapping \(m\).

Figure 4: Conditions for the success of launch.
may transfer control back to the caller via exit. pause models forced interruption by privileged software (e.g., device interrupt). 

**Attestation**: The attestation operator provided by the TAP ensures that the user is communicating with a bona fide enclave. The attestation operation can only be invoked from within the enclave and may be used by the enclave to establish its identity as part of an authentication protocol. At test returns a hardware-signed cryptographic digest of data and a measurement: . The signature uses the processor’s secret key , whose corresponding public key is signed by the trusted platform manufacturer.

### 4.2 The TAP Adversary

As described in § 2.2, the TAP adversary model is based on a privileged software attacker and consists of two components: (i) a tampering relation that describes how the adversary can change the platform’s state, and (ii) an observation function that describes what elements of machine state are visible to the adversary. The adversary model is parameterized and has three instantiations: Adversary MCP, adversary MC and adversary M, which are defined below. MCP is the most general adversary and models all capabilities of a privileged software attacker, while MC and M are restricted.

#### 4.2.1 Adversary Tampering

The tamper relation for the TAP formalizes all possible actions that may be carried out by a software attacker. It serves two purposes. When reasoning about integrity, the tamper relation defines all operations that an adversary may carry out to interfere with enclave execution. When reasoning about confidentiality, the tamper relation models computation performed by the adversary in order to learn an enclave’s private state. The most general tamper relation corresponds to Adversary MCP, and is defined as the adversary carrying out an unbounded number of the following actions:

1. Unconstrained updates to pc and regs. These are modeled using the havoc statement commonly used in program verification: havoc_pc, havoc_regs.
2. Loads and stores to memory with unconstrained address (va) and data (data) arguments.
   - \(\langle op, hitx \rangle \leftarrow fetch(va)\)
   - \(\langle regs[ri], hitx \rangle \leftarrow load(va)\)
   - \(hitx \leftarrow store(va, data)\)
3. Modification of the adversary’s view of memory by calling get_addr_map and set_addr_map with unconstrained arguments.
   - \(set_addr_map(e, v, p, perm)\)
   - \(regs[ri] \leftarrow get_addr_map(e, v)\)
4. The invocation of enclave operations with unconstrained arguments.
   - \(\text{Launch enclave: }\langle \text{launch}(e, m, x, p, t)\rangle\)
   - \(\text{Destroy enclave: }\langle \text{destroy}(e)\rangle\)
   - \(\text{Enter and resume enclave: }\langle \text{exit}(e)\rangle\)
   - \(\langle \text{Pause (exit) from and interrupt (pause) enclave}\rangle\)

Any adversary program, including malicious operating systems, hypervisors, malicious enclaves and privileged malware, can be modeled using the above actions. Therefore, this adversary model allows us to reason about TAP security guarantees in presence of a general software adversary.

**Restricted Adversaries**: Adversary MCP is restricted to computation based on memory values and cache state; it ignores the value returned by get_addr_map. Adversary M only computes using memory values; it ignores hitf, hiti and hitm returned by fetch, load and store, respectively, in addition to the result of get_addr_map.

#### 4.2.2 Adversary Observation Model

The observation function captures what state the user expects to be attacker-visible.

**Adversary M**: The observation function \(obs_M^\sigma(e)\) is a partial map from physical addresses to words and allows the adversary to observe the contents of all memory locations not private to enclave . It is defined as \(\sigma(mem[p])\) when \(\sigma(owner[p]) \neq e\) and \(\perp\) otherwise.

**Adversary MC**: The observation function \(obs_{MC}^\sigma\) specifies that besides contents of memory locations that are not private to an enclave, the adversary can also observe whether these locations are cached. It is also a partial map from physical addresses to words and is defined to be the tuple \(\langle \sigma(mem[p]), \text{cached}(\sigma, p)\rangle\) when \(\sigma(owner[p]) \neq e\) and \(\perp\) otherwise. \(\text{cached}(\sigma, p)\) is true iff physical address \(p\) stored in the cache in the machine state \(\sigma\).

Note that the adversary cannot directly observe whether an enclave’s private memory locations are cached. However, unless cache sets are partitioned between the attacker and the enclave, cache attacks [68, 74] allow the adversary to learn this information.
Adversary MCP: This observation function extends the adversary’s capabilities by allowing observation of the virtual to physical mappings and associated access/permission bits for each virtual address. It is defined as:

\[ \text{obs}_e^{\text{MCP}}(\sigma) = \lambda v. \; (\text{obs}_e^{\text{MC}}(\sigma), \lambda v. \; \sigma(\text{get_addr_map}(e, v))) \]

The notation \( \sigma(\text{get_addr_map}(e, v)) \) refers to the result of evaluating \( \text{get_addr_map}(e, v) \) in the state \( \sigma \).

4.2.3 Enclave and Adversary State and Inputs. Recall that the state of an enclave \( e \) is \( E_e(\sigma) \). \( E_e(\sigma) \) is defined as the tuple \( \langle E_{\text{mem}}(e, \sigma), E_{\text{regs}}(e, \sigma), E_{\text{pc}}(e, \sigma), E_{\text{cfg}}(e, \sigma) \rangle \) if \( e \in \text{enc_metadata} \) and ⊥ otherwise. The components of this tuple are as follows:

- \( E_{\text{mem}}(e, \sigma) \) is a partial map from virtual addresses to words. It is defined to be \( \sigma(\text{mem}[\text{enc_metadata}[e].addr_map[v][PA]]) \) if \( v \in \sigma(\text{enc_metadata}[e].evrange) \) and ⊥ otherwise. In other words, \( E_{\text{mem}} \) refers to the content of each virtual memory address in the enclave’s evrange.
- \( E_{\text{regs}}(e, \sigma) \) is \( \sigma(rregs) \) if \( \text{curr}(\sigma) = e \) (when the enclave is executing), and \( \sigma(\text{enc_metadata}[e].rregs) \) otherwise.
- \( E_{\text{pc}}(e, \sigma) \) is \( \sigma(ppc) \) if \( \text{curr}(\sigma) = e \) (when the enclave is executing), and \( \sigma(\text{enc_metadata}[e].ppc) \) otherwise.
- The tuple \( E_{\text{cfg}}(e, \sigma) \) consists of the following elements:
  1. \( \sigma(\text{enc_metadata}[e].addr_map) \)
  2. \( \sigma(\text{enc_metadata}[e].entrypoint) \)
  3. \( \sigma(\text{enc_metadata}[e].evrange) \)

Recall that the input to enclave \( e \) at state \( \sigma \) is \( I_e(\sigma) \), which is the tuple \( I_e(\sigma) = (I_e^P(\sigma), I_e^U(\sigma)) \). \( I_e^P(\sigma) \) is the random number provided to the state \( \sigma \). \( I_e^U(\sigma) \) is a partial map from virtual address to words. It is \( \sigma(\text{mem}[\text{enc_metadata}[e].addr_map[v][PA]]) \) if each of these conditions hold: (i) enclave \( e \) is executing: \( \text{curr}(\sigma) = e \), (ii) \( v \) is mapped to some physical address: \( \sigma(\text{valid}(\text{enc_metadata}[e].addr_map[v])) \), and (iii) \( v \) is not private: \( v \notin \sigma(\text{enc_metadata}[e].evrange) \); it is ⊥ otherwise. In other words, \( I_e^U(\sigma) \) refers to the contents of each virtual address not in the enclave’s evrange.

The output \( O_e(\sigma) \) contains memory values outside enclave’s evrange. \( O_e(\sigma) \) is defined identically to \( I_e^U(\sigma) \). The adversary’s state \( A_e(\sigma) \) is modeled as the tuple \((S_e(\sigma), \hat{E}_e(\sigma))\). \( S(\sigma) \) is \( \langle S_{\text{mem}}(\sigma), S_{\text{regs}}(\sigma), S_{\text{cfg}}(\sigma) \rangle \) denotes privileged software state. \( S_{\text{mem}}(\sigma) = \lambda v. \; \sigma(\text{mem}[\text{os_metadata}.addr_map[v][PA]]) \) is its view of memory, \( S_{\text{regs}}(\sigma) \) denotes the privileged software’s registers: \( \sigma(rregs) \) when privileged software is executing, and \( \sigma(\text{os_metadata}.rregs) \) otherwise. \( S_{\text{cfg}}(\sigma) \) is the privileged software’s virtual to physical mappings: \( \sigma(\text{os_metadata}.addr_map) \). \( \hat{E}_e(\sigma) \) is the state of all the other enclaves in the system except for enclave \( e \): \( \hat{E}_e(\sigma) = \alpha e'. \; \text{ite}(e \neq e', E_e(\sigma), \bot) \). (ite is if-then-else.)

4.3 Proof of Secure Remote Execution for TAP

We proved three machine-checked theorems that correspond to the requirements for secure remote execution as described in § 3.

TAP Integrity: We proved that the integrity result (Equation 4) holds for the TAP for all three adversaries: M, MC and MCP. This shows that these adversaries have no effect on enclave execution beyond providing inputs via non-enclave memory.

TAP Measurement: We showed that Equation 2 and Equation 3 are satisfied by the TAP. The proof for Equation 3 need not include adversarial operations because integrity ensures that an adversary cannot affect enclave’s execution beyond providing inputs.

TAP Confidentiality: We showed three confidentiality results, each corresponding to the three TAP adversaries: M, MC, and MCP.

Confidentiality holds unconditionally for adversary M.

For adversary MC, let \( p_{\text{set}} : \text{PA} \rightarrow \text{set} \) be the function that maps physical addresses to cache sets. This function is uninterpreted (abstract) in the TAP and will be defined by implementation.

We showed that confidentiality holds for adversary MC if Equation 6 is satisfied: a physical address belonging to an enclave never shares a cache set with a physical address outside the enclave.

\[ \forall p_1, p_2, s, e. \]
\[ \sigma(\text{owner}[p_1] = e \land \sigma(\text{owner}[p_2] \neq e) \implies (p_{\text{set}}(p_1) \neq p_{\text{set}}(p_2)) \]

Finally, we showed that confidentiality holds for adversary MCP if Equation 6 is satisfied by the TAP implementation and the TAP configuration Boolean priv mappings is true.

5 REFINEMENTS OF THE TAP

We prove that models of MIT Sanctum and Intel SGX are refinements of the TAP under certain adversarial parameters. Refinement shows that each operation, including all adversarial operations, on Sanctum and SGX processors can be mapped to an "equivalent" TAP action. The refinement proof implies SRE, which was proven on the TAP, also holds for our models of SGX and Sanctum.

5.1 Refinement Methodology

Let \( \text{Impl} = (\Sigma_L, \sim_L, \text{init}_L) \) be a transition system with states \( \Sigma_L \), transition relation \( \sim_L \) and initial state \( \text{init}_L \). We say that \( \text{Impl} \) refines TAP, or equivalently TAP simulates \( \text{Impl} \), if there exists a simulation relation \( R \subseteq (\Sigma_L \times \Sigma) \) with the following property:

\[ \left( \forall s_j \in \Sigma_L, s_k \in \Sigma_L, \sigma_j \in \Sigma. \right. \]
\[ \left( s_j, \sigma_j \right) \in R \land s_j \sim_L s_k \]
\[ \left. \left( (s_k, \sigma_k) \in R \lor (\exists \sigma_k \in \Sigma. \; \sigma_j \sim \sigma_k \land (s_k, \sigma_k) \in R) \right) \right\} \]
\[ \land \left( \text{init}_L \right) \in R \]

\[ \begin{array}{c}
\sigma_j \\
\sim_L \\
s_j \\
\sigma_k \\
\end{array} \\
\begin{array}{c}
R \\
s_k \\
\end{array} \]

Figure 5: Illustration of Stuttering Simulation.

The condition states that for every pair of states \( (s_j, \sigma_j) \) (belonging to \( \text{Impl} \) and TAP respectively) that are related by \( R \), if \( \text{Impl} \) steps from \( s_j \) to \( s_k \), then either (i) the TAP takes no steps, and \( s_k \) is related to \( \sigma_j \), or (ii) there exists a state \( \sigma_k \) of TAP such that \( \sigma_j \) steps to \( \sigma_k \) and \( (s_k, \sigma_k) \) are related according to \( R \). In addition, the initial states of \( \text{Impl} \) and TAP must be related by \( R \). This is illustrated in Figure 5. This corresponds to the notion of stuttering simulation [12],
and we require stuttering because a single invocation of launch corresponds to several API calls in Sanctum and SGX.

Refinement states that every trace of Impl can be mapped using relation \( R \) to a trace of TAP; effectively this means that TAP has a superset of the behaviors of Impl. The security properties of the TAP are hyperproperties, which in general, are not preserved by refinement [17]. However, the properties we consider are 2-safety properties [67] that are variants of observational determinism [45, 55]. These are properties are preserved by refinement. Intuitively, refinement asserts that any behavior observable on Impl has an equivalent behavior on TAP via \( R \), so any attacker advantage available on Impl but not on TAP precludes refinement. The SRE properties proven on the TAP therefore also hold for SGX and Sanctum, as we show that these architectures (conditionally) refine the TAP.

5.2 MIT Sanctum Processor
Sanctum [19] is an open-source enclave platform that provides strong confidentiality guarantees against privileged software attackers. In addition to protecting enclave memory from direct observation and tampering, Sanctum protects against software attackers that seek to observe an enclave’s memory access patterns.

5.2.1 Sanctum Overview. Sanctum implements enclaves via a combination of hardware extensions to RISC-V [5] and trusted software at the highest privilege level, called the security monitor. Sanctum Hardware: Sanctum minimally extends Rocket Chip [4, 5], an open source reference implementation of the RISC-V ISA [71, 72]. Specifically, Sanctum hardware isolates physical addresses by dividing system memory (DRAM) into regions, which use disjoint last level cache sets, and allocating each region to an enclave exclusively. Since enclaves have exclusive control over one or more DRAM regions, there is no leakage of private memory access patterns through the cache. An adversary cannot create a TLB entry that maps its virtual address to an enclave’s private cache set.

Sanctum Monitor: The bulk of Sanctum’s logic is implemented in a trusted security monitor. The monitor exclusively operates in RISC-V’s machine mode, the highest privilege-level implemented by the processor, and solely able to bypass virtual address translation. Monitor data structures maintain enclave and DRAM region state. The monitor configures the Sanctum hardware to enforce low-level invariants that comprise enclave access control policies. For example, the monitor places an enclave’s page tables within that enclave’s DRAM region, preventing the OS from monitoring an enclave’s page table metadata to infer memory access patterns. The monitor exposes an API for enclave operations, including measurement. A trusted bootloader boots the system, loads the monitor and creates a chain of certificates authenticating the Sanctum chip and the loaded security monitor.

5.2.2 Sanctum Model. Our Sanctum model combines the Sanctum hardware and the reference security monitor, and includes hardware registers, hardware operations, monitor data structures and the monitor API. The hardware registers include the DRAM bitmap which tracks ownership of DRAM regions, page table base pointers, and special regions of memory allocated to the monitor and for direct memory access (DMA). Hardware operations modeled include page tables and address translation, and memory instruction fetch, loads and stores. All monitor APIs are modeled.

5.2.3 Sanctum Model Refines TAP. The Sanctum refinement proof is expressed in three parts.

1. Concrete MMU refines Abstract MMU: We constructed an abstract model of a memory management unit (MMU). The abstract MMU’s address translation is similar to the TAP; it is a single-level map from virtual page numbers to permissions and physical page numbers. In contrast, the concrete Sanctum MMU models a multi-level page table walk in memory. We showed that the concrete Sanctum MMU model refines the abstract MMU. We then used the abstract MMU in modeling the Sanctum Monitor. This simplifies the simulation relation between Sanctum and TAP.

2. Sanctum Simulates TAP: We showed that every Sanctum state and every Sanctum operation, which includes both enclave and adversary operations, can be mapped to a corresponding TAP operation. For this, (i) we constructed a simulation relation between Sanctum states and corresponding TAP states, and (ii) we constructed a Skolem function [64] mapping each Sanctum operation to the corresponding TAP operation. We proved that for every pair of states in the simulation relation, every Sanctum operation can be mapped by the Skolem function to a corresponding TAP operation such that the resultant states are also in the simulation relation.

3. Proof of Cache Partitioning: The Sanctum model instantiates the function pa2set which maps physical addresses to cache sets. We showed that the Sanctum API’s init enclave operation and the definition of pa2set together ensure that all Sanctum enclaves’ cache sets are partitioned, i.e. Equation 6 is satisfied.

5.3 Intel SGX
Intel Software Guard Extensions extend the Intel architecture to enable execution of enclave programs.

5.3.1 SGX Overview. Unlike Sanctum, enclaves in SGX are implemented in hardware (microcode), which provides an instruction set extension [30] to create enclaves (ecreate), enter enclaves (enter), generate attested statements (ereport), etc. The SGX processor dedicates a contiguous region of physical memory (called the enclave page cache, or EPC), exclusively useable for storing enclave pages. While this provides confidentiality of enclave’s memory, SGX does not protect several side channel leaks such as cache timing or page access patterns. The last level cache is shared between the enclave and the adversary, and adversary memory can map to the same cache set as the enclave’s private address, allowing the attacker to perform cache attacks [11, 46, 68, 74]. SGX also allows the OS to examine the page tables that control the enclave’s private memory, enabling the OS to read the accessed and dirty bits, thus learning the enclave’s memory access pattern at the page-level granularity. The OS also gets notified on page fault exceptions (as part of demand paging), and this is another channel to learn the enclave’s page-level memory access patterns [61, 73].

5.3.2 SGX Model. Similar to Sanctum, we use a formal model of the SGX instruction set, following the approach of Moat [63], which models the SGX platform at the level of abstraction presented in the Intel Programmer Reference manual [30]. The model contains ISA-level semantics of the SGX instructions. It includes hardware
state such as enclave page cache metadata (EPCM), page tables, private core registers, etc. The model elides details such as attestation, hardware encryption of DRAM pages, and the cryptographic protections (encryption, integrity, and freshness) of demand paging and instead assumes several axioms about these features.

5.3.3 SGX Model Refines TAP We attempted to prove that SGX refines TAP, in that all SGX traces can be mapped to TAP traces. However, we cannot prove SGX refinement unconditionally. We show that refinement holds only when priv_mappings = false (see Sec. 4.1.2). This is because SGX implements a mechanism for the attacker (OS) to view page table entries, which contains the accessed and dirty bits. As TAP confidentiality for Adversary MCP only holds when priv_mappings = true, SGX is not secure against MCP. Furthermore, the lack of cache partitioning also prevents us from showing that Equation 6 holds, so SGX does not refine TAP instantiated with Adversary MC. We are able to prove refinement of TAP by SGX for the restricted adversary M. This shows SGX provides similar guarantees to Sanctum, except for leakage through the cache and page table side-channels.

6 VERIFICATION RESULTS

This section discusses our models and machine-checked proofs. Our models of the TAP, Intel SGX and MIT Sanctum are constructed using the BoogiePL [6, 22] intermediate verification language. BoogiePL programs can be annotated with assertions, pre-conditions and post-conditions for procedures and loop invariants. The validity of these annotations are checked using the Boogie verification condition generator [6], which in turn uses automated theorem provers like the Z3 SMT solver [21].

6.1 BoogiePL Model Construction

<table>
<thead>
<tr>
<th>Description</th>
<th>#pr</th>
<th>#fn</th>
<th>#an</th>
<th>#ln</th>
<th>Verif. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAP</td>
<td>22</td>
<td>49</td>
<td>204</td>
<td>1752</td>
<td>5</td>
</tr>
<tr>
<td>Integrity</td>
<td>12</td>
<td>13</td>
<td>145</td>
<td>985</td>
<td>26</td>
</tr>
<tr>
<td>Measurement</td>
<td>6</td>
<td>3</td>
<td>100</td>
<td>800</td>
<td>6</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>8</td>
<td>-</td>
<td>200</td>
<td>1388</td>
<td>194</td>
</tr>
<tr>
<td>MMU Model</td>
<td>9</td>
<td>13</td>
<td>68</td>
<td>739</td>
<td>7</td>
</tr>
<tr>
<td>MMU Refinement</td>
<td>3</td>
<td>2</td>
<td>38</td>
<td>216</td>
<td>8</td>
</tr>
<tr>
<td>Sanctum</td>
<td>23</td>
<td>321</td>
<td>44</td>
<td>780</td>
<td>1</td>
</tr>
<tr>
<td>Sanctum Refinement</td>
<td>12</td>
<td>3</td>
<td>94</td>
<td>548</td>
<td>11</td>
</tr>
<tr>
<td>SGX</td>
<td>36</td>
<td>113</td>
<td>4</td>
<td>1526</td>
<td>-</td>
</tr>
<tr>
<td>SGX Refinement</td>
<td>10</td>
<td>1</td>
<td>38</td>
<td>351</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>141</strong></td>
<td><strong>518</strong></td>
<td><strong>935</strong></td>
<td><strong>9085</strong></td>
<td><strong>260</strong></td>
</tr>
</tbody>
</table>

Table 4: BoogiePL Models and Verification Results

Table 4 shows the approximate size of each of our models. #pr, #fn, #an and #ln refers to the number of procedures, functions, annotations and lines of code respectively. Annotations refer to the number of loop invariants, assertions, assumptions, pre- and post-conditions that we manually specify. While Boogie can infer discharge some assertions automatically, we found that we had to manually specify 935 annotations before it accepted our proofs.

The rows TAP, MMU Model, Sanctum and SGX correspond to models of the functionality of the TAP, the Sanctum MMU, Sanctum, and SGX respectively. The other rows correspond to our proofs of SRE and refinement. In total, the models are about 4800 lines of code while the proofs form the remaining ≈ 4300 lines of code. A significant part of the effort in developing the proofs was finding the correct invariants to help Boogie prove the properties.

BoogiePL can only verify safety properties. But many of our theorems involve hyperproperties [17]. We used the self-composition construction [7, 67] to convert these into safety properties. BoogiePL is also incapable of verifying properties involving alternating nested quantifiers, for example, ∀x. ∃y. φ(x, y). We Skolemized [64] such properties to take the form: ∀x. ∃y. (y = f(x)) ⇒ φ(x, y); f(x) is called a Skolem function and must be manually specified.

6.2 Verification Results

Table 4 lists the total time taken by Boogie/Z3 to check validity of all the manually specified annotations — by verifying the annotations, we omit them from the trusted computing base, which only includes the Boogie/Z3 theorem prover. The verification times for the TAP, MMU Model and Sanctum rows is for proving that procedures in these models satisfy their post conditions, which specify behavior and system invariants. The verification times for the remaining rows is the time taken to prove the SRE properties and refinement. The total computation time in checking validity of the proofs once the correct annotations are specified is only a few minutes.

7 DISCUSSION

We argue that TAP-based verification of enclave platforms is more than just a set of proofs of correctness for Intel SGX and MIT Sanctum. TAP serves as a specification of primitives for enclave execution, and it is designed to be extensible towards additional features (e.g., non-volatile monotonic counters, demand paging) and additional guarantees against sophisticated attackers (e.g., timing attacks). While our proofs for secure measurement, integrity, and confidentiality will need to be modified to support these extensions, the proof skeletons will remain the same.

7.1 Implications for Enclave Platforms

The TAP can be beneficial to implementers of enclave platforms in the following ways. First, the TAP can be used as a top-down specification for what operations an enclave platform must support. Implementers can use refinement checks to ensure that the TAP’s security guarantees apply to the implementation as well. A important implementation challenge is that security is not compositional: addition of operations to a secure platform can make it insecure. The insecurity of SGX to Adversary MCP [61, 73] stems from demand paging, using which the OS observes the state of enclave’s page tables; this feature is not present in Sanctum. Suppose that the next version of Sanctum supports demand paging through the use of oblivious RAM to maintain TAP’s confidentiality guarantee. Reasoning about these more complex enclave platforms is infeasible without a verification methodology and TAP-like specification of the platform’s primitives.
Second, the TAP can also be used bottom-up, as this paper does, to reason about the security properties of existing platforms. Such analysis exposes differences between various platforms, e.g., leakage through the cache and page tables for SGX.

7.2 Implications for Enclave Software

While techniques for verifying security properties of enclave programs [62, 63] have been developed, they rely on models of the underlying enclave platform. We argue that such reasoning is better done using TAP’s clean abstraction which is simpler than the instruction-level model of SGX and API-level model of Sanctum. This simplification makes automated verification more scalable, yet retains soundness because of the refinement checks. It is also a step towards enabling portability among different enclave platforms.

Most importantly, this paper provides a common language for research into security properties of enclaves. While side-channel defenses have been proposed for SGX enclaves [51, 60, 61], the lack of formalization of the enclave’s execution, attacker’s operations and observations, and the desired confidentiality property makes it infeasible to systematically compare two defenses.

8 RELATED WORK

Secure Processors: There have been several commercial deployments of secure processors. ARM TrustZone [1] implements a secure and normal mode of execution to effectively create a single privileged enclave in an isolated address space. TPM+TXT [27] enables attestation of the platform’s state, but includes all privileged software layers in the trusted computing base. Most recently, Intel SGX [2, 18, 29, 44] implements unprivileged enclaves protecting the integrity of memory and enclave state against software adversaries and certain physical access attacks. A formal cryptographic model of SGX’s anonymous attestation scheme is presented in [54]. Academic work seeking to improve the security of aspects of conventional processors is also abundant [23, 40, 41, 50].

Several clean-slate academic projects have been seeking to build a trusted system. The XOM [39] architecture introduced the concept of isolated software containers managed by untrusted host software, and employed encryption and HMAC to protect DRAM. Aegis [66] showed how a security kernel could measure and sign enclaves, and employed a Merkle tree to guarantee freshness of data in DRAM. Bastion [15] encrypts and HMACs DRAM and employs a trusted (and authenticated as part of remote attestation) hypervisor which is invoked at each TLB miss to check address translation against a policy. Ascend [25] and Phantom [43] ensure privacy and integrity of all CPU memory accesses through a hardware ORAM primitive.

Attacks on Secure Processors: Secure systems often expose complex threat models in practice, leading to vulnerabilities in the application layers. Side channel observations, such as attacks observing cache timing, are known to compromise cryptographic keys used by numerous cryptosystems [10, 13, 14, 34]. Attacks exploiting other shared resources exist as well, such as those observing a core’s branch history buffers [35]. These attacks are viable at any privilege, separated by arbitrary protection boundaries [31, 42, 52]. Similar attacks apply on trusted hardware, as shown on SGX with attacks observing shared caches [11, 46, 59], and shared page tables [61, 73].

Non-Interference and Hyperproperties: The security properties of secure measurement, integrity, and confidentiality are formulated as 2-safety observational determinism properties [45, 55], which is a restricted class of hyperproperties [17]. SRE relates closely to the notion of non-interference introduced by Goguen and Meseguer [26], and separability proposed by Rushby [56]. Our confidentiality definition is an adaptation of standard non-interference to the enclave execution model. Separability provides isolation from other programs on the system, but it is too strict for practical applications as it forbids communication between programs and assumes the absence of covert channels; it also does not consider privileged software adversaries as the formalism assumes a safe underlying hardware-software system.

Security Type Systems: A large body of work has studied type systems that enforce information-flow security [16, 20, 58, 65, 70]. Recent examples for hardware design are [37, 38, 75]. SecVerilog [75] extends the Verilog hardware description language with dependent type annotations that define a security policy statically verified by the SecVerilog compiler. One could conceivably implement SGX or Sanctum processors using SecVerilog, thus guaranteeing the implementation do not have unsafe information flow. However, these techniques reason about the security policy at the level of individual signals in the hardware. A higher level of abstraction (like TAP) is needed for reasoning about enclave software.

Machine-Checked Proofs of Systems: We perform machine-checked proofs in this work, and similar efforts have verified other classes of systems. The seL4 project [33, 47] proves isolation and information flow enforcement in the seL4 microkernel using the Isabelle/HOL proof assistant [49]. The Ironclad project [28] built a fully verified software stack (including an OS, device drivers, and cryptographic libraries) from the ground-up, but this approach is unlikely to scale to real-world OS and system software. On the other hand, enclaves are small, trusted components of an application, and therefore more amenable to formal analysis. The miTLS project [9] is building a verified reference implementation of TLS which complements our work nicely — enclaves indubitably require TLS channels to communicate with other enclaves and clients. Vijayaraghavan et al. [69] used the Coq proof assistant [8] to verify correctness of a multiprocessor directory-based cache coherence protocol. While our Boogie [6, 22] proofs do involve manual effort, we contend that they are more automated than their hypothetical counterparts in systems such as Isabelle and Coq.

9 CONCLUSION

This paper introduced a framework and methodology to reason about the security guarantees provided by enclave platforms. We introduced the Trusted Abstract Platform (TAP), and performed proofs demonstrating that TAP satisfies the three properties required for secure remote execution (SRE): secure measurement, integrity and confidentiality. We then presented machine-checked proofs stating that models of Intel SGX and Sanctum are refinements of TAP under certain adversarial conditions. Therefore, these platforms also satisfy the properties required for SRE. Overall, this paper took a step towards a unified, extensible framework for reasoning about enclave programs and platforms.
Table 5: Glossary of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{N} )</td>
<td>Natural numbers.</td>
</tr>
<tr>
<td>( \mathbb{B} )</td>
<td>Booleans ( (\mathbb{B} = {\text{true}, \text{false}}) ).</td>
</tr>
<tr>
<td>( \lambda x. \ expr )</td>
<td>Function with argument ( x ); computes ( expr ).</td>
</tr>
<tr>
<td>( m[i] )</td>
<td>Element ( i ) in map ( m ).</td>
</tr>
<tr>
<td>( \text{ite}(c, x, y) )</td>
<td>Evaluates to ( x ) if ( c ) is \text{true}, ( y ) otherwise.</td>
</tr>
<tr>
<td>( \text{rec.} f1d )</td>
<td>Field ( f1d ) in record ( rec ).</td>
</tr>
<tr>
<td>VA</td>
<td>Virtual addresses.</td>
</tr>
<tr>
<td>PA</td>
<td>Physical addresses.</td>
</tr>
<tr>
<td>ACL</td>
<td>Permissions for virt. addr. (R/W/X etc.)</td>
</tr>
<tr>
<td>Tag</td>
<td>Cache tags.</td>
</tr>
<tr>
<td>( E_{id} )</td>
<td>Type of enclave “ids” (integer/pointer).</td>
</tr>
<tr>
<td>( E_M )</td>
<td>Enclave metadata type.</td>
</tr>
<tr>
<td>( \Sigma )</td>
<td>Set of all TAP states.</td>
</tr>
<tr>
<td>( \sim )</td>
<td>TAP transition relation.</td>
</tr>
<tr>
<td>init</td>
<td>TAP initial state.</td>
</tr>
<tr>
<td>( \sigma, \sigma_0, \sigma_1, \ldots )</td>
<td>TAP states. ( (\sigma \in \Sigma) ).</td>
</tr>
<tr>
<td>( \sigma_i \sim \sigma_j )</td>
<td>TAP steps from state ( \sigma_j ) to ( \sigma_j ).</td>
</tr>
<tr>
<td>( \pi, \pi_0, \pi_1, \ldots )</td>
<td>Traces of the TAP.</td>
</tr>
<tr>
<td>( \pi[0] )</td>
<td>Initial state of trace ( \pi ).</td>
</tr>
<tr>
<td>( \pi[i] )</td>
<td>( i )th state of in trace ( \pi ).</td>
</tr>
<tr>
<td>( \sigma(expr) )</td>
<td>Expression ( expr ) evaluated in state ( \sigma ).</td>
</tr>
<tr>
<td>( e, e_1, e_2, \ldots )</td>
<td>Enclave programs.</td>
</tr>
<tr>
<td>( E_e(\sigma) )</td>
<td>Enclave ( e )'s state when in platform state ( \sigma ).</td>
</tr>
<tr>
<td>( I_e(\sigma) )</td>
<td>Enclave ( e )'s input when in platform state ( \sigma ).</td>
</tr>
<tr>
<td>( R_e(\sigma) )</td>
<td>Randomness component of enclave ( e )'s input.</td>
</tr>
<tr>
<td>( U_e(\sigma) )</td>
<td>Untrusted component of enclave ( e )'s input.</td>
</tr>
<tr>
<td>( O_e(\sigma) )</td>
<td>Enclave ( e )'s output when in platform state ( \sigma ).</td>
</tr>
<tr>
<td>( A_e(\sigma) )</td>
<td>Adversary's state when in platform state ( \sigma ).</td>
</tr>
<tr>
<td>( I^p(\sigma) )</td>
<td>Non-deterministic component of platform state.</td>
</tr>
<tr>
<td>( \mu(e) )</td>
<td>Measurement of enclave ( e ).</td>
</tr>
<tr>
<td>( \lfloor e \rfloor )</td>
<td>Set of all traces of ( e ).</td>
</tr>
<tr>
<td>( m[v]_{PA} )</td>
<td>Physical address for virtual address ( v ) in virtual to physical mapping ( m ).</td>
</tr>
<tr>
<td>valid ( (m[v]) )</td>
<td>Predicate that corresponds to whether virtual address ( v ) is mapped to some physical address in the virtual to physical mapping ( m ).</td>
</tr>
<tr>
<td>executable ( (m[v]) )</td>
<td>Predicate that corresponds to whether virtual address ( v ) has execute permissions in the virtual to physical mapping ( m ).</td>
</tr>
<tr>
<td>( \Sigma_L )</td>
<td>Set of all implementation states.</td>
</tr>
<tr>
<td>( \sim_L )</td>
<td>Implementation transition relation.</td>
</tr>
<tr>
<td>init ( _L )</td>
<td>Implementation initial state.</td>
</tr>
<tr>
<td>( s, s_0, s_1, \ldots )</td>
<td>Implementation states. ( (s \in \Sigma_L) ).</td>
</tr>
<tr>
<td>( s_i \sim_L s_j )</td>
<td>Implementation steps from state ( s_j ) to ( s_j ).</td>
</tr>
<tr>
<td>( m_1</td>
<td></td>
</tr>
<tr>
<td>PK ( k )</td>
<td>Public key ( k ).</td>
</tr>
<tr>
<td>SK ( k )</td>
<td>Secret key ( k ).</td>
</tr>
<tr>
<td>( {m}_{SK_k} )</td>
<td>Msg ( m ) encrypted/signed with key ( SK_k ).</td>
</tr>
</tbody>
</table>