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  
ilbebedev@mit.edu

Srinivas Devadas  
Massachusetts Institute of Technology  
devadas@mit.edu

Sanjit A. Seshia  
University of California, Berkeley  
sseshia@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), a formalization of idealized enclave platforms along with a parameterized adversary. We also formalize the notion of secure remote execution and present machine-checked proofs showing that the TAP satisfies the three key security properties that entail 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.

CCS CONCEPTS
• Security and privacy → Formal methods and theory of security; Security in hardware; Trusted computing; Information flow control;

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

1 INTRODUCTION

An application executing on a typical computing platform contains large privileged software layers in its trusted computing base (TCB), which includes the operating system (OS), hypervisor and firmware. Vulnerabilities in these privileged software layers have been exploited to enable the execution of privileged malware with disastrous consequences [1, 26, 35, 52, 64]. To address this problem, processor designers and vendors are now developing CPUs with hardware primitives, such as Intel SGX and MIT Sanctum, which isolate sensitive code and data within protected memory regions, called enclaves, that are inaccessible to other software running on the machine. With these primitives, enclaves become the only trusted components of an application, and a carefully programmed enclave can execute safely even when the privileged software layers are compromised.

Despite growing interest, there has only been informal security analysis of these new platforms, which we refer to as 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 [55, 68, 69]. However, comparing the security guarantees and adversary models (of say T-SGX [68] and Shinde et al. [69]) 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 [13] 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

CCS ’17, October 30–November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of CCS ’17, https://doi.org/10.1145/3133956.3134099.
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 model of a privileged software attacker with varying 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 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 the notion of 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 a triad of integrity, confidentiality, and secure measurement properties, which are more easily formally verified 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 [75]. These models are designed to be 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 (with hash-based measurement $μ(e)$) 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 evrange which maps the enclave’s protected memory (i.e., each address in evrange either maps to a physical address that is owned by the enclave or is inaccessible due to page permissions) and (3) permissions for each address within evrange.

Enclave Programs: The user ships an enclave $e = (\text{init}_e, \text{config}_e)$. config_e defines the enclave’s entrypoint: config_e.entrypoint, its virtual address range: config_e.evrange, and its access permissions: config_e.acl. init_e specifies the enclave’s initial state at launch, including values of code and data pages within config_e.evrange.

We assume that the enclave’s virtual address range and corresponding access permissions (config_e.evrange, config_e.acl) are the same for local and remote executions. The assumption is required in order to ensure that enclave measurement (see § 3.1.1) returns a predictable value.\(^3\)

Enclave State: At any point of time, the machine is in some state $s$, which we leave abstract for now and defer its definition to § 4.1. The enclave’s state $E_e(s)$ is a projection of the machine state, and specifies a valuation of the following state variables: (1) memory vmem : VA → W which is a partial map from virtual addresses (within config_e.evrange) to machine words, (2) general-purpose registers regs : $\mathbb{N} → W$ which are indexed by a natural number, (3) program counter pc : VA, and (4) configuration config_e, which is copied verbatim from $e$ and remains constant throughout the enclave’s execution. init_e specifies the state of enclave’s memory (vmem) at the time of launch. We abuse notation to also write init_e(E_e(s)) to mean that $E_e(s)$ is in its launch-time state (prior to first entry into the enclave).

Enclave Inputs and Outputs: In addition to the private state $E_e(s)$, an enclave accesses non-enclave memory for reading inputs and writing outputs, both of which are under adversary control — since addresses outside evrange can be read and modified by both the adversary and the enclave, we assume reads from these addresses return unconstrained values. The enclave may also invoke the enclave platform’s primitive to get random numbers (which we also treat as inputs). Therefore, we define enclave’s input $I_e(s) = (I^P_e(s), I^U_e(s))$, where $I^P_e(s)$ is any random number that is provided by the platform in that step ($e$ if randomness was not requested), and $I^U_e(s)$ is the projection of the machine state that $e$ may read and the attacker may write; specifically, $I^U_e(s)$ specifies an evaluation of non-enclave memory: a partial map from virtual addresses (outside config_e.evrange) to machine words. Similarly, we define enclave’s output $O_e(s)$ to be a projection of the machine state that $e$ can write and the attacker may read; specifically, $O_e(s)$ specifies an evaluation of non-enclave memory: a partial map from virtual addresses (outside config_e.evrange) to words. Furthermore, an attacker runs concurrently with $e$ and maintains state that cannot be accessed by the enclave (e.g., hypervisor private memory, other malicious enclaves), which we denote as $A_e(s)$.

Enclave Execution: Our semantics of enclave execution assumes that an enclave program is deterministic modulo the input $I_e(s)$, i.e., the next state of the enclave is a function of the current state $E_e(s)$ and input $I_e(s)$ (which includes $I^P_e(s)$ and $I^U_e(s)$). This is not a restriction in practice as both Sanctum and SGX enclaves interact with the external world via memory-based I/O ($I^P_e(s)$), and besides the random bits ($I^U_e(s)$) from an entropy source, there are no other sources of non-determinism — we only assume single threaded enclaves in this work. Since we make this deterministic assumption while defining an enclave’s semantics, we must prove that the platform does not allow non-determinism.

The enclave computes by performing steps, where in each step 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 $A_e(s)$. This model of execution lets us define the platform’s transition relation $\sim$, where $(\sigma_1, \sigma_2) \in \sim$ indicates that the platform can transition from $\sigma_1$ to $\sigma_2$; from hereon, we write this in infix form as $\sigma_1 \sim_\sim \sigma_2$. 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 $I^P(s)$, which is only available to the privileged software — the enclave’s source of non-determinism is captured in $I_e(s)$. We require that the platform be deterministic relative to $I^P(s)$. A secure platform must also ensure an enclave program $e$ is deterministic relative to its input $I_e(s)$. 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 $A_e(s)$, enclave’s input $I^U_e(s)$ for any enclave $e$, and launch or destroy any number of enclaves. We model the adversary’s effect through the tamper relation over pairs of states: $(\sigma_1, \sigma_2) \in \text{tamper}$ 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 ∋ e 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: ∀σ. (σ, σ) ∈ tamper. 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_e and configuration config_e.

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(σ) denote the result of an observation for the machine state σ. For instance, an attacker that only observes outputs enjoys the observation function obs = (O_e(σ)). 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 π = ⟨σ_0, σ_1, . . . , σ_m⟩, such that ∀i. σ_i ∋ σ_i+1; π[i] refers to the ith element of the trace. Since the attacker may pause and resume e at any time, we define e’s execution to be the subsequence of states from π where e is executing. To that end, let the function curr(σ) denote the current mode of the platform, where curr(σ) = e iff the platform executes enclave e in state σ. Using this function, we can filter out the steps in π where e is not executing. We write the resulting sequence as ⟨σ_0', σ_1', . . . , σ_m'⟩ such that init_e(E_e(σ_0')) ∧ ∀i. curr(σ_i') = e. This subsequence is the enclave’s execution trace: ((I_e(σ_0'), E_e(σ_0'), O_e(σ_0'))), . . . , (I_e(σ_m'), E_e(σ_m'), O_e(σ_m'))). 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 ∀i. (σ_i', σ_i'+1) ∈ tamper. This has the effect of favoring A_e(σ) and I_e(σ) 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.

\[ [e] = \{ (I_e(σ_0'), E_e(σ_0'), O_e(σ_0'), . . . ) \mid \text{init}_e(E_e(σ_0)) \} \tag{1} \]

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 deterministic property of enclave programs, a specific sequence of inputs (I_e(σ_0'), I_e(σ_1'), . . . , I_e(σ_m')) 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 enclave 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 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 attestations 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 traces 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 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 program provides SRE to any enclave program. We provide machine-checked proofs of SRE for the TAP in § 4, and show how this applies to models of Intel SGX and MIT Sanctum 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 obs.

3.1.1 Secure Measurement. During launch, the platform computes a hash of the enclave’s initial contents (init) along with relevant configuration bits (config). The hash-based measurement acts as a unique identity for the enclave, which follows directly from the
We must prove it here.∀ operations may differ in the two traces. States and have the same input sequence will produce the same instructions, has no effect on the enclave’s execution. Any two traces In other words, the attacker only impacts an enclave’s execution if the two enclaves is the same, then the two enclaves execute identically in the same number of steps), (iii) if the input sequence to the two enclaves start with the same initial state for enclaves, (iii) if the input sequence to the two enclaves start with the same initial state for enclaves 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, has 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.

∀π₁, π₂. ∀i. (curr(π₁[i]) = e) ⇐⇒ (curr(π₂[i]) = e) (4)

∀i. (Eₑ(π₁[i]) = Eₑ(π₂[i]) ∧ Oₑ(π₁[i]) = Oₑ(π₂[i]))

Figure 2 shows the two traces from the integrity property. The adversary’s steps are labelled A₁ and A₂ 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 ≈E symbol. The initial state of the two enclaves is assumed to be the same and this is shown by the ≈ symbol. The adversary’s actions are Defined by the tamper function (see § 2.2.1), and these actions may differ between the two traces. The integrity proof must show that the enclave’s state and outputs do not differ despite this: ∀i. Eₑ(π₁[i]) = Eₑ(π₂[i]) ∧ Oₑ(π₁[i]) = Oₑ(π₂[i]). These proof obligations are denoted by the red ≈E and ≈O symbols. We assume that the adversary executes for the same number of steps in both traces. This does not restrict the adversary’s power as any attack that requires the adversary to execute for a different number of steps in the two traces can be simulated in our model by padding the adversary’s shorter trace with the appropriate number of "no-ops."

The theorem states that, given the above assumptions, enclave state and outputs are identical in the two traces at every step: ∀i. Eₑ(π₁[i]) = Eₑ(π₂[i]) ∧ Oₑ(π₁[i]) = Oₑ(π₂[i]).

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 a platform allows the attacker to observe different values in two such traces, we say that such a platform does not provide confidentiality.

Formally, to prove that an attacker learns no other information beyond obs, we must prove that for any two traces that have equivalent attacker operations and equivalent observations of the enclave execution, but possibly different enclave private states and executions, the attacker’s execution (its sequence of states) is identical.
∀π₁, π₂.

(Äₑ(π₁[0]) = Äₑ(π₂[0])
 ∧
 ∀i. curr(πᵢ[1]) = curr(πᵢ[2]) ∧ Iᵢ(πᵢ[1]) = Iᵢ(πᵢ[2])
 ∧
 ∀i. curr(πᵢ[1]) = e =⇒ obsₑ₁(πᵢ[1] + 1) = obsₑ₂(πᵢ[2] + 1))

Figure 3: Confidentiality property

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 enclave 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 gᵃ to the enclave, where x is a randomly generated fresh value.
   (b) Enclave sends its public parameter at test(gᵇ) 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 gᵃᵇ, and derive symmetric session keys sₑᵘ and sₑᵤ, a key for each direction.
3. The user now sends encrypted input to enclave using this shared secret: {in}sbₑ.
4. The enclave decrypts its input, performs the computation and returns the encrypted result to the user: {out}sbₑ.

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}sbₑ 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}sbₑ and any tampering to {in}sbₑ 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 [57] 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: 
\[ TAP = (\Sigma, \leadsto, init) \]
\( \Sigma \) is the set of states, \( \leadsto \) is the transition relation and \( init \in \Sigma \) is the initial state.

4.1.1 TAP State Variables. The states \( \Sigma \) of the TAP are defined as a valuation of the state variables \( Vars \). These variables are described in Table 1. pc, regs, mem have their usual meanings. addr_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 models a cache to show that confidentiality is preserved in the presence of a software adversary attempting cache attacks. The cache model assumes a physically-indexed and physically-tagged cache with symbolic parameters: an unbounded number of sets, unbounded associativity, and an arbitrary deterministic replacement policy. 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 applies to a broad space of set associative caches, and only requires that the cache set and cache tag for each memory access be deterministic functions of the physical address. The exact functions and parameters are specified by implementations (refinements) of TAP, which are models of SGX and Sanctum in this paper.

The variable current_eid tracks the enclave currently being executed. owner maps each physical address to the enclave which exclusively "owns" it. If 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 as the enclave itself. Attempts to access physical address \( p \) by all other enclaves and privileged software are blocked. owner[\( p \)] = OS means that address \( p \) is not allocated to any enclave. owner corresponds to the EPCM in SGX and the DRAM bitmap in Sanctum. It is the primary mechanism to enforce isolation of enclave’s private memory. enc_metadata stores metadata about each initialized enclave.

<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 → W</td>
<td>Architectural registers: map from natural numbers to words.</td>
</tr>
<tr>
<td>mem</td>
<td>PA → W</td>
<td>The memory: a map from physical addresses to words.</td>
</tr>
<tr>
<td>addr_map</td>
<td>VA → (ACL × PA)</td>
<td>Map from virtual addresses to permissions and physical addresses for current process.</td>
</tr>
<tr>
<td>cache</td>
<td>(Set × Way) → (( \beta ) × Tag)</td>
<td>Cache: map from a tuple of cache sets and ways to valid bits and cache tags.</td>
</tr>
<tr>
<td>current_eid</td>
<td>( \mathcal{E}_{id} )</td>
<td>Current enclave. current_eid = OS means that no enclave is being executed.</td>
</tr>
<tr>
<td>owner</td>
<td>PA → ( \mathcal{E}_{id} )</td>
<td>Map from physical address to the enclave address is allocated to.</td>
</tr>
<tr>
<td>enc_metadata</td>
<td>( \mathcal{E}<em>{id} \rightarrow \mathcal{E}</em>{M} )</td>
<td>Map from enclave ids to metadata record type (( \mathcal{E}_{M} )).</td>
</tr>
<tr>
<td>os_metadata</td>
<td>( \mathcal{E}_{M} )</td>
<td>Record that stores a checkpoint of privileged software state.</td>
</tr>
</tbody>
</table>

Table 1: Description of TAP State Variables

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.

<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>

Table 2: Fields of the enc_metadata record
We introduce a “setting” in the TAP, called
These conditions have subtle interactions with the requirements
points to under the mapping
Note:
Enclave Destruction: An enclave is deallocated using
attacks emphasizes the need for formal modeling and verification.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(v)</td>
<td>Fetch/read/write from/to virtual address v. Fail if v is not executable/readable/writeable respectively according to the addr_map or if owner[addr_map[v], PA] ≠ current_eid.</td>
</tr>
<tr>
<td>load(v)</td>
<td>Get/set virtual to physical mapping and associated permissions for virtual address v.</td>
</tr>
<tr>
<td>store(v)</td>
<td>Initialize enclave e by allocating enc_metadata[e].</td>
</tr>
<tr>
<td>get_addr_map(e, v)</td>
<td>Set mem[p] to 0 for each p such that owner[p] = e. Deallocate enclave enc_metadata[e].</td>
</tr>
<tr>
<td>set_addr_map(e, v, p, perm)</td>
<td>enter(e, resume(e)</td>
</tr>
<tr>
<td></td>
<td>Enter enclave, pause also saves a checkpoint of pc and regs and sets enc_metadata[e].paused = true.</td>
</tr>
<tr>
<td>exit(t), pause()</td>
<td>Return hardware-signed message with operand d and enclave measurement e: {d</td>
</tr>
</tbody>
</table>

Table 3: Description of TAP Operations

\[
\begin{align*}
\text{(current_eid = OS} ∧ e ≠ enc\_metadata & \land \\
\text{executable}[m[t]] ∧ t ∈ x_o, } & \land \\
\forall p. p ∈ x_p ⇒ \text{owner}[p] = OS & \land \\
\forall v. u ∈ x_o, ⇒ (\text{valid}(m[v]) \iff m[v]_PA ∈ x_p) & \land \\
\forall v_1, v_2, (v_1 ∈ x_o ∧ v_2 ∈ x_o, ) ⇒ (m[v_1]_PA \neq m[v_2]_PA) & ⇐ (\text{launch_status} = \text{success}) 
\end{align*}
\]

Note: m[v]_PA refers to physical address that virtual address v points to under the mapping m.

Figure 4: Conditions for the success of launch

permit privileged software to access an enclave’s private page tables. We introduce a “setting” in the TAP, called priv_mappings, and this insecure behavior is allowed when priv_mappings = false.

Enclave Creation: The launch(e, m, x_o, x_p, t) operation is used to create an enclave. The enclave’s virtual to physical address mapping and associated permissions are specified by m. x_o is the set of enclave-private virtual addresses (evrange). It corresponds to the base address and size arguments passed to ECREATE in SGX and create_enclave in Sanctum. x_p is the set of physical addresses allocated to the enclave and its entrypoint is the virtual address t.

The launch operation only succeeds if enclave id e does not already exist, if the entrypoint is mapped to an enclave-private executable address, if every virtual address in x_o that is accessible to the entrypoint is a physical address in x_p, and if there is no aliasing among the addresses in x_o. A precise statement of the conditions that result in a successful launch shown in Figure 4. These conditions have subtle interactions with the requirements for SRE. For example, if virtual addresses within x_o are allowed to alias, an adversary can construct two enclaves which have the same measurement but different semantics. The potential for such attacks emphasizes the need for formal modeling and verification.

Enclave Destruction: An enclave is deallocated using destroy, which zeroes out the enclave’s memory so that its private state is not leaked when the privileged software claims the memory. This is necessary for confidentiality because untrusted privileged software can destroy an enclave at any time.

Enclave Entry/Exit: The operation enter(e) enters enclave e by setting the pc to its entrypoint and current_eid to e. resume continues execution from a checkpoint saved by pause. The enclave 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. attest returns a hardware-signed cryptographic digest of data d and a measurement: d || μ(e) || {d || μ(e)}_SK_p. The signature uses the processor’s secret key SK_p, 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 enclave 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, regs.
2. Loads and stores to memory with unconstrained address (va) and data (data) arguments.
• \(<op, hit_f> \leftarrow \text{fetch}(va)\)
• \(<\text{regs}[r], hit_t> \leftarrow \text{load}(va)\)
• \(hit_t \leftarrow \text{store}(va, data)\)

(3) Modification of the adversary’s view of memory by calling \(\text{set\_addr\_map}\) and \(\text{get\_addr\_map}\) with unconstrained arguments.
• \(\text{set\_addr\_map}(e, v, p, \text{perm})\)
• \(\text{regs}[r] \leftarrow \text{get\_addr\_map}(e, v)\)

(4) The invocation of enclave operations with unconstrained arguments.
• Launch enclaves: \(\text{launch}(e, m, x_c, x_p, t)\).
• Destroy enclaves: \(\text{destroy}(e)\).
• Enter and resume enclaves: \(\text{enter}(e)\) and \(\text{resume}(e)\).
• Exit (exit) from and interrupt (pause) enclaves.

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 \(MC\) is restricted to computation based on memory values and cache state; it ignores the value returned by \(\text{get\_addr\_map}\). Adversary \(M\) only computes using memory values; it ignores \(hit_f, hit_t\) and \(hit_m\) returned by \(\text{fetch}, \text{load}\) and \(\text{store}\), respectively, in addition to the result of \(\text{get\_addr\_map}\).

4.2.2 Adversary Observation Model. The observation function captures what state the user expects to be attacker-visible.

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

\[ \text{obs}_M^E(\sigma) \equiv \phi_4(\sigma) \equiv \lambda p. \text{rtx}(\sigma(\text{owner}[p]) \neq e, \sigma(\text{mem}[p]), \bot) \]

**Adversary \(MC\):** The observation function \(\text{obs}_C^MC(\sigma)\) 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 \((\sigma(\text{mem}[p]), \text{cached}(\sigma, p))\) when \(\sigma(\text{owner}[p]) \neq e\) and \(\bot\) otherwise. \(\text{cached}(\sigma, p)\) is true iff physical address \(p\) stored in the cache in the machine state \(\sigma\).

\[ \text{obs}_C^MC(\sigma) \equiv \lambda p. \text{rtx}(\sigma(\text{owner}[p]) \neq e, \sigma(\text{mem}[p]), \text{cached}(\sigma, p), \bot) \]

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 [78,84] allow the adversary to learn this information.

**Adversary \(MCP\):** The MCP adversary’s capabilities are extended beyond \(MC\) to include observation of the virtual to physical mappings and associated access permission bits for each virtual address.

\[ \text{obs}_C^{MCP}(\sigma) \equiv \lambda v. \left(\text{obs}_C^MC(\sigma), \lambda v. \sigma(\text{get\_addr\_map}(e, v))\right) \]

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 \(E_e(\sigma)\), which 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 \(\bot\) 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 \(\sigma(\text{mem[enc\_metadata]}[e, \text{addr\_map}][\text{v\_PA}])\) if \(v \in \sigma(\text{enc\_metadata}[e, \text{addr\_map}])[\text{v\_PA}]\) and \(\bot\) otherwise. In other words, \(E_{\text{mem}}\) refers to the content of each virtual memory address in the enclave’s \(\text{evrange}\).

• \(E_{\text{regs}}(e, \sigma)\) is \(\sigma(\text{regs})\) when \(e\) is executing, and \(\sigma(\text{enc\_metadata}[e, \text{regs}])\) otherwise.

• \(E_{\text{pc}}(e, \sigma)\) is \(\sigma(\text{pc})\) when \(e\) is executing, and \(\sigma(\text{enc\_metadata}[e, \text{pc}])\) otherwise.

• The tuple \(E_{\text{cfg}}(e, \sigma)\) consists of the following elements:
  (i) \(\sigma(\text{enc\_metadata}[e, \text{addr\_map}])\)
  (ii) \(\sigma(\text{enc\_metadata}[e, \text{entrypoint}])\)
  (iii) \(\sigma(\text{enc\_metadata}[e, \text{evrange}])\)

Recall that the input to enclave \(e\) at state \(\sigma\) is \(I_e(\sigma)\), which is the tuple \(I_e(\sigma) \equiv \langle I^E_{\text{cfg}}(\sigma), I^L_{\text{addr}}(\sigma)\rangle\). \(I^E_{\text{cfg}}(\sigma)\) is the random number provided at the state \(\sigma\). \(I^L_{\text{addr}}(\sigma)\) is a partial map from virtual address to words. It is \(\sigma(\text{mem[enc\_metadata]}[e, \text{addr\_map}][\text{v\_PA}])\), if each of these conditions hold: (i) enclave \(e\) is executing: \(\sigma(\text{pc}) = e\), (ii) \(v\) is mapped to some physical address: \(\sigma(\text{valid}(\text{enc\_metadata}.\text{addr\_map}[v]))\), and (iii) \(v\) is not private: \(v \notin \sigma(\text{enc\_metadata}[e, \text{evrange}])\), it is \(\bot\) otherwise. In other words, \(I^L_{\text{addr}}(\sigma)\) refers to the contents of each virtual address not in the enclave’s \(\text{evrange}\).

The output \(O_e(\sigma)\) contains memory values outside enclave’s \(\text{evrange}\). \(O_e(\sigma)\) is defined identically to \(I^L_{\text{addr}}(\sigma)\). The adversary’s state \(A_e(\sigma)\) is modeled as the tuple \(\langle S(e), \hat{E}_e(\sigma)\rangle\). \(S(e) \equiv \langle S_{\text{mem}}(e), S_{\text{regs}}(\sigma), S_{\text{cfg}}(\sigma)\rangle\) denotes privileged software state. \(S_{\text{mem}}(e)\) and \(\lambda v. \sigma(\text{mem[os\_metadata].addr\_map}[v][\text{v\_PA}])\) is its view of memory. \(S_{\text{cfg}}(\sigma)\) denotes the privileged software’s registers: \(\sigma(\text{regs})\) when privileged software is executing, and \(\sigma(\text{os\_metadata}.\text{regs})\) otherwise. \(S_{\text{cfg}}(\sigma)\) is the privileged software’s virtual to physical mappings: \(\sigma(\text{os\_metadata}.\text{addr\_map})\). \(\hat{E}_e(\sigma)\) is the state of all the other enclaves in the system except for enclave \(e\): \(\hat{E}_e(\sigma) \equiv \lambda e’. \text{rtx}(e' \neq e’, \hat{E}_e(\sigma), \bot)\), where \(\text{rtx}\) denotes 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 \(\text{pa2set} : \text{PA} \rightarrow \text{Set}\) be the function that maps physical addresses to cache sets. This function is interpreted (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
We prove that models of MIT Sanctum and Intel SGX are refinements of the TAP action. The refinement proof implies SRE, which was proven for the TAP. The security properties of the TAP are hyperproperties, which in general, are not preserved by refinement. However, the properties we consider are 2-safety properties that are variants of observational determinism. These 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 [21] 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 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 [81, 82]. 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 bootstraps 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 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 [72] 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 [33] to create enclaves (create), enter enclaves (enter), generate attested statements (report), etc. The SGX processor dedicates a contiguous region of physical memory (called the enclave page cache, or EPC), exclusively usable 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 [12, 49, 78, 84]. 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 [69, 83].

5.3.2 SGX Model. Similar to Sanctum, we use a formal model of the SGX instruction set, following the approach of Moat [71], which models the SGX platform at the level of abstraction presented in the Intel Programmer Reference manual [33]. 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 MCP. 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 [7, 24] intermediate verification language. BoogiePL programs can be annotated with assertions, loop invariants and pre-/post-conditions for procedures. The validity of these annotations are checked using the Boogie Verification Condition Generator [7], which in turn uses the Z3 SMT solver [23].

Procedure pre-/post-conditions enable modular verification using Boogie. For example, we specify the behavior of each TAP operation using pre- and post-conditions and verify that the implementation of these procedures satisfies these post-conditions. Once the post-conditions are verified, the TAP’s proofs of integrity, confidentiality and secure measurement can soundly ignore the implementation and only reason about the pre- and post-conditions of these operations.

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>
</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 inductive invariants to help Boogie prove the properties.

BoogiePL can only verify safety properties. But many of our theorems involve hyperproperties [19]. We used the self-composition construction [8, 77] 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 [72] such properties to take the form: ∀x. ∃y. (y = f(x)) \implies 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 our models of Intel SGX and MIT Sanctum. The TAP serves as a specification of primitives for enclave execution, and 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). It provides an extensible verification methodology that enables rigorous reasoning about the security properties of future enclave platforms. It is also an extensible model for reasoning about enclave programs.

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. An 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 [69, 83] stems from demand paging, using which the OS observes the state of enclave’s page tables; this feature is not present in the current implementation of 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. The refinement methodology described in this paper can be used to show that implementations of enclave platforms (e.g., the Sanctum implementation) are refinements of corresponding models (e.g., Sanctum model). Since we have shown that the Sanctum model is a refinement of the TAP, this proof would mean that the security properties of the TAP also hold for the Sanctum implementation.

7.2 Implications for Enclave Software

While techniques for verifying security properties of enclave programs [70, 71] 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 [55, 68, 69], 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.

7.3 Limitations

7.3.1 Limitations of the TAP. The TAP does not model the full space of systems for which enclaves are relevant. Instead, it focuses on enclave platforms that provide memory-based isolation. The TAP’s current adversary model assumes platform memory (DRAM) is trusted: memory can only be modified via software. Extending the adversary model beyond this to include, for example, memory probing/DMA attacks would require augmenting the TAP’s memory operations (fetch/load/store) with encryption and integrity protection. The TAP also does not support demand paging as a naive implementation is vulnerable to the pigeonhole attack [69, 83]. One path towards confidentiality-preserving demand paging in the TAP would be to use an oblivious-RAM protocol.

The TAP model and proofs are currently limited to concurrent execution on a single-threaded single-core processor. A fully general proof showing that the TAP ensures SRE on multicore systems would require parameterized verification of these properties for an unbounded number of cores, using for example, the CMP method [18]. We would also need to show linearizability of operations in the Sanctum/SGX models [31]. In this context, linearizability means that each operation appears to be atomic and there exists an ordering of operations across all cores that is consistent with their sequential orders of completion in individual cores. Linearizability of a model of SGX was shown by Leslie-Hurd et al. [39]. Extending the TAP to model security of simultaneously multithreaded (SMT) processors would require modeling many more side channels (e.g., branch predictors, shared resources such
as instruction queues and function units) and the development of architectural mechanisms that prevent leakage through these side channels.

Finally, the TAP does not model the cryptography in the attestation operation. Instead, we model the hash function using uninterpreted functions and assume the properties of collision resistance and second pre-image resistance.

### 7.3.2 Limitations of the Sanctum/SGX Models.

Our SGX model does not include demand paging, memory encryption or memory integrity protection and assumes correctness of SGX measurement. Unlike the TAP, the Sanctum implementation’s measurement is not computed on a single snapshot of state, but is instead updated incrementally during enclave creation. A proof that this incremental measurement is identical to the TAP’s measurement is ongoing. Like the TAP, our Sanctum model also uses uninterpreted functions to model the cryptography in the measurement operation.

### 8 RELATED WORK

#### Secure Processors

There have been several commercial deployments of secure processors. ARM TrustZone [2] implements a secure and normal mode of execution to effectively create a single privileged enclave in an isolated address space. TPM-TXT [29] enables attestation of the platform’s state, but includes all privileged software layers in the trusted computing base. Most recently, Intel SGX [3, 32, 47] implements unprivileged enclaves protecting the integrity of memory and enclave state against software adversaries and certain physical access attacks. Academic work seeking to improve the security of aspects of conventional processors is also abundant [24, 43, 44].

Several clean-slate academic projects have been seeking to build a trusted system. The XOM [42] architecture introduced the concept of isolated software containers managed by untrusted host software, and employed encryption and HMAC to protect DRAM. Aegis [76] showed how a security kernel could measure and sign enclaves, and employed counter-mode memory encryption and a Merkle tree to guarantee freshness of data in DRAM. Bastion [16] 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. Fides [74] uses a small trusted hypervisor to provide security guarantees for so-called protected modules; protected modules offer very similar programming abstractions and security guarantees as enclaves. Sancus [54] builds upon Fides, and proposes hardware extensions that ensure security of protected modules in the context of resource-constrained embedded processors. Ascend [27] and Phantom [46] 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 [11, 14, 15, 37]. Attacks exploiting other shared resources exist as well, such as those observing a core’s branch history buffers [38]. These attacks are viable at any privilege, separated by arbitrary protection boundaries [34, 45, 56]. Similar attacks apply on trusted hardware, as shown on SGX with attacks observing shared caches [12, 49, 66], and shared page tables [69, 83].

#### Formal Models/Verification of Enclave Platforms

A formal cryptographic model of SGX’s anonymous attestation scheme is presented in [58]. Barbosa et al. [6] present a formal analysis of cryptographic protocols for secure outsourced computation using enclave platforms. In contrast to these efforts, which reason about cryptographic protocols in the context of enclaves, our work formalizes the security guarantees of enclave platforms themselves. Patrignani et al. [60, 61] develop abstract trace semantics for low-level code (not including side-channels) on protected module architectures – Fides [74] and Sancus [54] – in order to build secure compilers for these platforms [59]. This is complementary to our work which focuses on formalizing the security guarantees of enclave platforms in presence of a precisely modelled adversary (including side-channel observations).

While our work analyzes models of enclave platforms, verifying actual implementations remains an important challenge. One aspect of this was studied by Leslie-Hurd et al. [39]. They verified linearity of concurrent updates to shared SGX data structures.

**Non-Interference and Hyperproperties:** The security properties of secure measurement, integrity, and confidentiality are formulated as 2-safety observational determinism properties [48, 62], which is a restricted class of hyperproperties [19]. SRE relates closely to the notion of non-interference introduced by Goguen and Meseguer [28], and separability proposed by Rushby [63]. 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 [17, 22, 65, 73, 80]. Recent examples for hardware design are [40, 41, 85]. SecVerilog [85] 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 does 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. Software fault isolation prevents an untrusted software module from arbitrarily tampering the system’s state, and instead restrict all communication to occur via a narrow interface. RockSalt [50] uses Coq to reason about an x86 processor model and guarantee software fault isolation of an untrusted module.

**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 [36, 51] proves isolation and information flow enforcement in the seL4 microkernel using the Isabelle/HOL proof assistant [53]. The Ironclad project [30] built a fully verified software stack (including an OS, device drivers, and cryptographic libraries) from the ground-up. The mTLS project [10]
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. [79] used the Coq proof assistant [9] to verify correctness of a multiprocessor directory-based cache coherence protocol. While our Boogie [7, 24] 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.

ACKNOWLEDGMENTS

Funding for this research was partially provided by the National Science Foundation under grants CNS-1413920 and CNS-1528108, by the Intel ADEPT Center, by Delta Electronics, and by TerraSwarm, one of six centers of STARNet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] Lenovo ThinkPad System Management Mode Arbitrary Code Execution 0day Exploit. Available at https://github.com/Cr4sh/ThinkPwn.git.
[35] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. https://github.com/Cr4sh/ThinkPwn.git.
[38] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. CoRR, abs/1611.06952,
### 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>If-then-else operator: evaluates to (x) if (c) is true, (y) otherwise.</td>
</tr>
<tr>
<td>(\text{rec}.f)</td>
<td>Field (f) 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 virtual address (Readable/Writeable/Executable etc.)</td>
</tr>
<tr>
<td>Tag</td>
<td>Cache tags.</td>
</tr>
<tr>
<td>(\mathcal{E}_{\text{id}})</td>
<td>Type of enclave &quot;ids&quot; (integer/pointer).</td>
</tr>
<tr>
<td>(\mathcal{E}_{\text{M}})</td>
<td>Enclave metadata type.</td>
</tr>
<tr>
<td>(\Sigma)</td>
<td>Set of all TAP states.</td>
</tr>
<tr>
<td>(\rightarrow)</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 \rightarrow \sigma_j)</td>
<td>TAP steps from state (\sigma_i) 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>(\mathcal{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>(\mathcal{R}(\sigma))</td>
<td>Randomness component of enclave (e)'s input.</td>
</tr>
<tr>
<td>(\mathcal{U}(\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>(\mathcal{A}_e(\sigma))</td>
<td>Adversary's state when in platform state (\sigma).</td>
</tr>
<tr>
<td>(\mathcal{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>([e])</td>
<td>Set of all traces of (e).</td>
</tr>
<tr>
<td>(m[v]_{m,A})</td>
<td>Physical address for virtual address (v) in virtual to physical mapping (m).</td>
</tr>
<tr>
<td>valid((m[v]))</td>
<td>Is 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>Does 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>(\rightarrow_L)</td>
<td>Implementation transition relation.</td>
</tr>
<tr>
<td>init</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 \rightarrow_L s_j)</td>
<td>Implementation steps from state (s_i) 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>