A Formal Foundation for Secure Remote Execution of Enclaves

Pramod Subramanyan, University of California, Berkeley
Rohit Sinha, University of California, Berkeley
Ilia Lebedev, Massachusetts Institute of Technology
Srinivas Devadas, Massachusetts Institute of Technology
Sanjit A. Seshia, University of California, Berkeley

Presented by: ***Riccardo Paccagnella***

Motivations

- Lack of formalization in enclave platforms:
  - Developers cannot formally reason about the security of their programs.
  - Hardware designers cannot formally state the security properties of their architectures.
  - It is hard to compare and contrast potential improvements to these platforms.

Goals

1. Formalize a model of enclave execution in the presence of a software adversary and define the properties required for *secure remote execution* (SRE) of enclaves.

2. Introduce the *trusted abstract platform* (TAP), which serves as a specification of primitives for enclave execution and prove that it satisfies the SRE properties.

3. Prove that formal models of *Intel SGX* and *MIT Sanctum* are *refinements* of the TAP.

Formal model of enclave execution

Enclave platform: implements primitives to create enclaves.

Enclave program: $e = (init_e, config_e)$. $config_e$ defines entrypoint, virtual address range and access permissions; $init_e$ specifies initial state.

Enclave state: $E_e(\sigma)$ specifies a valuation of the state variables.

Enclave inputs: $I_e(\sigma) \doteq \langle I_e^R(\sigma), I_e^U(\sigma)\rangle$ where $I_e^R(\sigma)$ is any random number requested by the enclave and $I_e^U(\sigma)$ is an evaluation of the non-enclave memory that *e* may read and the attacker may write.

Enclave outputs: $O_e(\sigma)$ is an evaluation of the non-enclave memory that e may write and the attacker may read.

Formal model of enclave execution

The enclave is a finite state transition system, where $\sigma_i \rightsquigarrow \sigma_{i+1}$ indicates that the platform can transition from σ_i to σ_{i+1}.

An execution trace is a sequence of states denoted $\pi = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$, such that $\forall i\ \sigma_i \rightsquigarrow \sigma_{i+1}$.

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. All traces start in the initial state.

Formal model of the adversary

An enclave can execute in the presence of a privilege adversary. The adversary model consists of:

- Adversary tampering: pause the enclave; execute instructions that change the enclave's input; launch and destroy enclaves.

- Adversary observations: observe the contents of memory locations not private to the enclave; observe if these locations are cached; observe the virtual to physical mappings for each virtual address.

Secure remote execution of enclaves

Definition: 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*.

To formally verify that an enclave platform provides SRE to an enclave program, we decompose the SRE property into:

- <u>Secure measurement</u>: the platform must measure the enclave program to allow the user to detect any changes to it prior to execution.
- <u>Integrity</u>: the enclave's program execution cannot be affected by an attacker beyond providing inputs.
- <u>Confidentiality</u>: an attacker cannot distinguish between the execution of two enclaves, besides what is already revealed by obs.

Proof of secure measurement

Let *μ(e)* be the measurement of enclave *e*, computed when launching the enclave. We need to prove that:

1. Two enclaves with the same measurement have identical initial states.

$$\forall \sigma_1, \sigma_2.\ init_{e_1}(E_{e_1}(\sigma_1)) \wedge init_{e_2}(E_{e_2}(\sigma_2)) \Rightarrow \\ \mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2) \qquad (2)$$

2. If two enclaves *e₁* and *e₂* have the same state, then they produce equivalent execution traces for equivalent input sequences.

$$\begin{aligned} &\forall \pi_1, \pi_2. && (3)\\ &\Big(E_{e_1}(\pi_1[0]) = E_{e_2}(\pi_2[0]) && \wedge\\ &\forall i.\ (curr(\pi_1[i]) = e_1) \iff (curr(\pi_2[i]) = e_2) && \wedge\\ &\forall i.\ (curr(\pi_1[i]) = e_1) \implies I_{e_1}(\pi_1[i]) = I_{e_2}(\pi_2[i])\Big) && \implies\\ &\Big(\forall i.\ E_{e_1}(\pi_1[i]) = E_{e_2}(\pi_2[i]) \wedge O_{e_1}(\pi_1[i]) = O_{e_2}(\pi_2[i])\Big) \end{aligned}$$

Proof of integrity

We need to prove that:

1. 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.

$$
\begin{aligned}
&\forall \pi_1, \pi_2. && (4)\\
&\Big(E_e(\pi_1[0]) = E_e(\pi_2[0]) && \wedge\\
&\forall i.\ (curr(\pi_1[i]) = e) \iff (curr(\pi_2[i]) = e) && \wedge\\
&\forall i.\ (curr(\pi_1[i]) = e) \implies I_e(\pi_1[i]) = I_e(\pi_2[i])\Big) && \implies\\
&\Big(\forall i.\ E_e(\pi_1[i]) = E_e(\pi_2[i]) \wedge O_e(\pi_1[i]) = O_e(\pi_2[i])\Big)
\end{aligned}
$$

Proof of confidentiality

We need to prove that:

1. 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.

$$
\begin{aligned}
&\forall \pi_1, \pi_2. && (5)\\
&\Big(A_{e_1}(\pi_1[0]) = A_{e_2}(\pi_2[0]) && \wedge\\
&\forall i.\ curr(\pi_1[i]) = curr(\pi_2[i]) \wedge I^P(\pi_1[i]) = I^P(\pi_2[i]) && \wedge\\
&\forall i.\ curr(\pi_1[i]) = e \implies obs_{e_1}(\pi_1[i+1]) = obs_{e_2}(\pi_2[i+1])\Big) \implies &&\\
&\Big(\forall i.\ A_{e_1}(\pi_1[i]) = A_{e_2}(\pi_2[i])\Big) &&
\end{aligned}
$$

Goals

1. Formalize a model of enclave execution in the presence of a software adversary and define the properties required for *secure remote execution* (SRE) of enclaves.

2. Introduce the *trusted abstract platform* (TAP), which serves as a specification of primitives for enclave execution and prove that it satisfies the SRE properties.

3. Prove that formal models of *Intel SGX* and *MIT Sanctum* are *refinements* of the TAP.

The Trusted Abstract Platform (TAP)

The TAP is an idealization of an enclave platform. Formally, $TAP = (\Sigma, \rightarrow, init)$, where Σ is the set of states, $\rightarrow$ is the transition relation and $init \in \Sigma$ is the initial state.

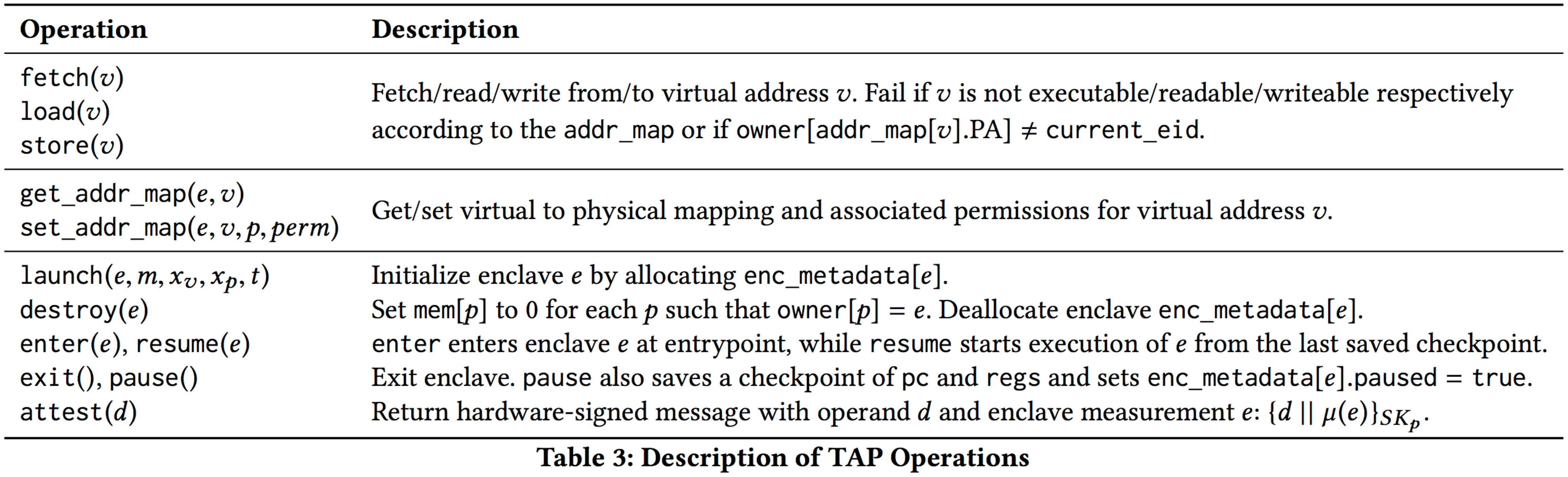
The states Σ of the TAP are defined as a valuation of the TAP state variables (Table 1).

State Var.	**Type**	**Description**
`pc`	VA	The program counter.
`regs`	$\mathbb{N} \rightarrow \mathrm{W}$	Architectural registers: map from natural numbers to words.
`mem`	$\mathrm{PA} \rightarrow \mathrm{W}$	The memory: a map from physical addresses to words.
`addr_map`	$\mathrm{VA} \rightarrow (\mathrm{ACL} \times \mathrm{PA})$	Map from virtual addresses to permissions and physical addresses for current process.
`cache`	$(\mathrm{Set} \times \mathrm{Way}) \rightarrow (\mathbb{B} \times \mathrm{Tag})$	Cache: map from a tuple of cache sets and ways to valid bits and cache tags.
`current_eid`	$\mathcal{E}_{id}$	Current enclave. `current_eid` = $\mathcal{OS}$ means that no enclave is being executed.
`owner`	$\mathrm{PA} \rightarrow \mathcal{E}_{id}$	Map from physical address to the enclave address is allocated to.
`enc_metadata`	$\mathcal{E}_{id} \rightarrow \mathcal{E}_{\mathcal{M}}$	Map from enclave ids to metadata record type ($\mathcal{E}_{\mathcal{M}}$).
`os_metadata`	$\mathcal{E}_{\mathcal{M}}$	Record that stores a checkpoint of privileged software state.

Table 1: Description of TAP State Variables

The Trusted Abstract Platform (TAP)

Table 3 describes the operations supported by the TAP.



Operation	Description
`fetch`(v) `load`(v) `store`(v)	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`]` $\neq$ `current_eid`.
`get_addr_map`(e, v) `set_addr_map`($e, v, p, perm$)	Get/set virtual to physical mapping and associated permissions for virtual address v.
`launch`(e, m, x_v, x_p, t) `destroy`(e) `enter`(e), `resume`(e) `exit`(), `pause`() `attest`(d)	Initialize enclave e by allocating `enc_metadata[`e`]`. Set `mem[`p`]` to 0 for each p such that `owner[`p`]` = e. Deallocate enclave `enc_metadata[`e`]`. `enter` enters enclave e at entrypoint, while `resume` starts execution of e from the last saved checkpoint. Exit enclave. `pause` also saves a checkpoint of `pc` and `regs` and sets `enc_metadata[`e`]`.paused = `true`. Return hardware-signed message with operand d and enclave measurement e: $\{d \mid\mid \mu(e)\}_{SK_p}$.

Table 3: Description of TAP Operations

TAP satisfies the SRE requirements

Three machine-checked theorems that correspond to the requirements for secure remote execution were proved with a BoogiePL model.

Goals

1. Formalize a model of enclave execution in the presence of a software adversary and define the properties required for *secure remote execution* (SRE) of enclaves.

2. Introduce the *trusted abstract platform* (TAP), which serves as a specification of primitives for enclave execution and prove that it satisfies the SRE properties.

3. Prove that formal models of *Intel SGX* and *MIT Sanctum* are *refinements* of the TAP.

Refinements of the TAP

Let *Impl* = $\langle \Sigma_L, \rightsquigarrow_L, init_L \rangle$ be a transition system with states Σ_L, transition relation $\rightsquigarrow_L$ and initial state $init_L$. We say that *Impl* refines *TAP* if there exists a simulation relation $R \subseteq (\Sigma_L \times \Sigma)$ with the following property:

$$
\begin{aligned}
&\Big(\forall s_j \in \Sigma_L, s_k \in \Sigma_L, \sigma_j \in \Sigma. && (7)\\
&\ (s_j, \sigma_j) \in R \land s_j \rightsquigarrow_L s_k && \implies\\
&\ \Big((s_k, \sigma_j) \in R \lor (\exists \sigma_k \in \Sigma.\ \sigma_j \rightsquigarrow \sigma_k \ \land\ (s_k, \sigma_k) \in R)\Big)\Big) && \land\\
&(init, init_L) \in R
\end{aligned}
$$

Refinement methodology

1. Develop a formal model of the *Impl* enclave platform.

2. Attempt to prove that all *Impl* traces can be mapped to TAP traces.

Result: Unlike Sanctum, SGX does not refine TAP with a privileged software adversary model. This is because SGX implements a mechanism for the OS to view page table entries, which contain the accessed and dirty bits.

SGX only refines a version of TAP which leaks some side channel observations to the attacker, therefore providing a weaker confidentiality guarantee.

Goals

1. Formalize a model of enclave execution in the presence of a software adversary and define the properties required for *secure remote execution* (SRE) of enclaves.

2. Introduce the *trusted abstract platform* (TAP), which serves as a specification of primitives for enclave execution and prove that it satisfies the SRE properties.

3. Prove that formal models of *Intel SGX* and *MIT Sanctum* are *refinements* of the TAP.

Limitations of the TAP

- TAP focuses only on enclave platforms that provide memory-based isolation.
- The TAP model and proofs are currently limited to a single-threaded, single-core execution.
- TAP does not model the cryptography in the attestation operation.
- The adversary model of TAP only focuses on privileged <u>software</u> attackers.

Discussion

- TAP enables rigorous reasoning about the security properties of <u>enclave platforms</u>.
  - It can be used as a top-down specification for what operations an enclave platform must support.
  - It can be used bottom-up, to reason about the security properties of existing platforms.
- TAP provides a common language for research into security properties of enclaves.
- TAP's clean abstraction is a step forward enabling portability among different enclave platforms.