Better Trust Zone: Verifying Security of Enclave-Aware Calculi

Aaron Bembrnek  Lily Tsai  Ezra Zigmond
Harvard University
bembenek@g.harvard.edu  lilliantsai@college.harvard.edu  ezigmond@college.harvard.edu

Abstract

Techniques from language-based security, such as security type systems, often provide protection against high-level adversaries but cannot guarantee security against low-level attackers, such as those that can inject code or inspect memory. We model in Coq a security-typed calculus that includes an abstract model of enclaves, and show that it can provide meaningful security guarantees, even in the face of low-level attackers. We also model a translation scheme from an enclave-agnostic calculus to an enclave-aware calculus and show that the scheme ensures that enclaves are correctly placed, so that a translation of a well-typed program in the enclave-agnostic calculus results in a well-typed program in the enclave-aware calculus.

1. Introduction

Programs that are well-typed under a security-type system provably enforce security properties against high-level attackers, such as ensuring that the program does not leak sensitive information through either programmer error or malicious inputs provided by an adversarial user [11]. However, these guarantees often do not hold in the presence of low-level attackers like those that can arbitrarily inject code or inspect memory. We formalize in Coq a security-type system that can provide meaningful security guarantees in the face of these type of attackers by using an abstract model of enclaves, a hardware mechanism for isolating code and data.

Our formalization is heavily based on the work of Gollamudi and Chong [6], who propose a security-typed enclave-aware calculus (IMPE) and prove that it is secure against purely passive attackers who observe program execution and in certain cases, attackers who can modify both enclave and non-enclave code. IMPE provides security against such low-level attackers by placing sensitive code and data in enclave memory. This models Intel SGX-style enclaves [2], which provide strong hardware-enforced isolation, to the extent that not even privileged code such as the operating system can inspect or modify memory or execution within an enclave. We model a simplified version of IMPE (SIMPE) in Coq and provide a machine-checkable proof that a well-typed program in SIMPE is secure against both passive and active (code-modifying) attacker models.

Because manual placement of enclaves is potentially tedious and unintuitive for programmers, Gollamudi and Chong also propose a constraint-based translation scheme from IMPS, a calculus that is not enclave-aware, to IMPE. They prove that any translation that meets the constraints of this scheme will, given a well-typed IMPS program, produce a well-typed IMPE program that enforces the security policies of the source program. Since IMPE programs are secure against low-level attackers, this translation scheme enables programmers to write programs with low-level security guarantees without being burdened by details of low-level hardware mechanisms. We model this translation scheme in Coq and verify that the constraints are in fact sufficient for ensuring that a translation will produce a well-typed IMPE program given a well-typed IMPS program.

To summarize, we make the following contributions:

• Coq models of the security-typed enclave-aware calculus (IMPE) from Gollamudi and Chong and SIMPE, a simplified variant of IMPE (Section 2);
• a Coq proof that a well-typed program in SIMPE is secure in both a passive and an active (code-modifying) attacker model (Section 3);

\footnote{Our code can be found at https://github.com/aaronbembenek/verified-auto-enclave.}
• a discussion of our experience translating pen-and-paper proof techniques common in the programming language literature to a machine-checkable environment (Sections 4); and

• Coq models of the security-typed calculus without enclaves (IMPS) and the translation scheme from IMPS to IMPE proposed by Gollamudi and Chong, as well as a Coq proof that the translation scheme preserves well-typedness (Section 5).

2. Enclave-Aware Calculus Models

We present an enclave-aware calculus, IMPE, based on the calculus introduced by Gollamudi and Chong [6] and SIMPE, a simplified version of IMPE for which we verify security properties. Both calculi model important features of enclaves and support information security policies. We first describe IMPE and then explain how SIMPE simplifies the IMPE model.

The IMPE model can be broken down into three components: IMPE security policies, IMPE syntax and semantics, and the typing judgments that define a well-typed IMPE program.

2.1 Security Policies

Security policies in IMPE support a set of security levels \( \mathcal{L} = \{L, H, \top\} \). Level \( L \) corresponds to low-security information that anyone, including an attacker, is allowed to learn. Security level \( H \) is used for high-security, privileged information that an attacker should not be able to learn. Finally, security level \( \top \) is used to label information that no one, privileged or otherwise, is allowed to learns.

A security policy is either a security level policy \( \ell \) where \( \ell \in \mathcal{L} \) or an erasure policy \( \ell_1 \not\rightarrow \text{end} \ell_2 \) where \( \ell_1, \ell_2 \in \mathcal{L} \). A security level policy represents a static policy where information is always at a particular security level, while an erasure policy indicates that the security level of information will change after a condition \( \text{cond} \) is met. For example, the erasure policy \( H \not\rightarrow \text{done} \top \) means that information is privileged until condition \( \text{done} \) is set, at which point it must not be anywhere in the system.

2.2 Syntax and Semantics

IMPE supports standard imperative language constructs such as conditional statements, while loops, command sequences, variable assignments, memory updates, and function calls. Figure 1 shows the inductive types \( \text{exp} \) and \( \text{com} \) for IMPE expressions and commands.

All observable effects of IMPE are produced using the output command, which outputs to either the \( L \) or \( H \) security channel. Furthermore, the only input to an IMPE program is the initial memory, so all of a program’s sensitive information is contained in its initial memory.

IMPE adds support for two security features: escape hatches and erasure policies. Intuitively, an escape hatch is a computation over high-security memory whose result can be output to a low-security channel. IMPE allows for escape hatches by providing a declassify command. Erasure policies are implemented using IMPE conditions and condition com-

```
Inductive mode : Type :=
| None : mode
| Encl : enclave \rightarrow mode.

Inductive exp : Type :=
| Enat : nat \rightarrow exp
| Evar : var \rightarrow exp
| Ebinop : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}
| Eloc : location \rightarrow \text{exp}
| Ederef : \text{exp} \rightarrow \text{exp}
| Eisunset : \text{condition} \rightarrow \text{exp}
| Elambda : \text{mode} \rightarrow \text{com} \rightarrow \text{exp}
with com : Type :=
| Cskip : com
| Cassign : var \rightarrow \text{exp} \rightarrow \text{com}
| Cdeclassify : var \rightarrow \text{exp} \rightarrow \text{com}
| Cupdate : \text{exp} \rightarrow \text{exp} \rightarrow \text{com}
| Coutput : \text{exp} \rightarrow \text{sec_level} \rightarrow \text{com}
| Ccall : \text{exp} \rightarrow \text{com}
| Cset : \text{condition} \rightarrow \text{com}
| Cenclave : \text{enclave} \rightarrow \text{com} \rightarrow \text{com}
| Ckill : \text{enclave} \rightarrow \text{com}
| Cseq : \text{list com} \rightarrow \text{com}
| Cif : \text{exp} \rightarrow \text{com} \rightarrow \text{com} \rightarrow \text{com}
| Cwhile : \text{exp} \rightarrow \text{com} \rightarrow \text{com}.

Inductive val : Type :=
| Vlambda : \text{mode} \rightarrow \text{com} \rightarrow \text{val}
| Vnat : nat \rightarrow \text{val}
| Vloc : location \rightarrow \text{val}.

Definition mem : location \rightarrow \text{val}.
Definition register : variable \rightarrow \text{val}.
```

Figure 1: IMPE syntax.
mands. The isunset(cnd) expression and set(cnd) command are used to read and write to conditions.

To model enclaves, IMPE adds the enclave(i,c) command, the kill(i) command, and provides execution modes. The enclave(i,c) command executes the command c inside of enclave i. This corresponds to an enclave enter instruction provided by hardware which causes the processor to execute the code in an enclave [2]. The only way to enter enclave i is through the enclave(i,c) command. The kill(i) command kills enclave i, and the code and data in a killed enclave can never be accessed.

A mode is used to tag memory locations with the enclave to which they belong, if any. A mode can either be an enclave Encl i or None. A mode-specifying function δ: location → mode maps memory locations to the enclave (if any) to which they belong. Function expressions (Elambda) are parametrized by modes, which represents the mode of memory in which the function resides. Entering an enclave through the enclave(i,c) command changes the execution mode to Encl i. The IMPE semantics model isolation properties of enclaves by enforcing that any memory locations belonging to an enclave are only readable or writable while executing in that particular enclave mode.

The semantics for IMPE are defined using a large-step operational semantics. The evaluation judgment takes an expression or command configuration and produces a final configuration. An IMPE command configuration specifies the command, register state, memory, and set of killed enclaves. An IMPE expression configuration is the same as a command configuration, but contains an expression instead of a command. A final configuration indicates the register state, memory, and set of killed enclaves after the command has finished executing. It also includes a trace containing the observable outputs produced during execution, namely those values produced by output commands.

Figure 2 gives the definitions for IMPE configurations and semantic judgments. The semantic judgment is parameterized by a mode of execution, which indicates which enclave, if any, the command is executing in. The judgment is also parameterized by the global function δ, which is fixed at the beginning of the program’s execution. These parameters are used in the semantics to ensure that code executing in non-enclave mode None cannot dereference some memory location

\begin{verbatim}
Definition exp_config :=
  exp * register * mem * set enclave
Definition com_config :=
  com * register * mem * set enclave
Definition final_config :=
  register * mem * set enclave
Definition trace := list outputs

Parameter δ: location → mode.
Inductive exp_step : mode → exp.config →
  value → Prop
Inductive com_step : mode → com.config →
  final.config → trace → Prop
\end{verbatim}

Figure 2: IMPE semantic judgments.

\end{verbatim}

\[ δ(\text{loc}) = \text{Encl } i \text{ for some } i. \]

IMPE’s semantic judgment for expressions is similar to that for commands. Instead of producing a final configuration, the expression evaluates to a value. Evaluating an expression does not modify any registers or memory, kill enclaves, or produce any observable output.

All IMPE programs begin execution in mode None, with an initial register state reg_init that maps all variables to the value 0.

2.3 Type System

Every IMPE expression has a given security-type \( σ_p \), which includes both the basic type \( σ \) and a security policy \( p \). The variable environment context \( Γ \) and the location environment context \( Σ \) track the types of values in registers and at memory locations, respectively.

\( Γ \) is updated with the appropriate types when register variables are assigned values in the program. \( Σ \) is fixed at the beginning of the program’s execution: every location is assigned a fixed security-type, and any value at location \( \text{loc} \) must have the type \( Σ(\text{loc}) \). Neither \( Γ \) nor \( Σ \) can ever map to a type with security level \( ⊤ \). Because \( Σ \) never changes, we do not include it as an argument of the typing judgment.

Figure 3 shows the inductive types for expression and command typing judgments.

The typing judgment \( \text{com.wt} \) takes as an argument a set of conditions that are unset when the command \( \text{com} \) is executed. This is necessary to determine the current security level of each location, since a location may have an erasure security policy.
The IMPE type system is designed to provide security guarantees about any well-typed IMPE program. First, no high-security values can be directly exposed to a low-security channel. Furthermore, information about high-security values cannot be leaked through implicit flows. The type system tracks implicit flows with a program counter security level. All commands are typed with a given program counter \( pc \), and the type system ensures that the command cannot output values to a channel with a security level lower than \( pc \). Any branch of a conditional command (\( \text{Cwhile}\) or \( \text{Cif}\) ) must be well-typed with a program counter \( pc_b \), that is at least as secure as the program counter of the conditional command. In addition, \( pc_b \) must be at a higher security level than the expression that decides the branch of the conditional: if the expression evaluates to a value typed with security level \( p \), \( p \) must be less than \( pc_b \). Figure 4 shows how the typing judgment for a \( \text{Cwhile}\) command ensures that the program counter \( pc_b \) of the command inside the conditional is at least as secure as both \( p \) and \( pc \).

Figure 5 shows an example of a program that is not well-typed because the type system prevents implicit flows. Suppose that \( \Sigma(\text{password}) = T\text{nat}_H \) and \( \Sigma(\text{guess}) = T\text{nat}_H \). The expression \( \text{password} == \text{guess} \) decides the branch of the conditional. This expression evaluates to a value typed with security level \( H \) because the computation is performed over locations with security level \( H \). The output commands performed by each branch are therefore typed with a program counter \( H \), and are prevented from outputting to the low channel by the type system.

The type system also adds enclave-specific guarantees: any high security values must be stored in enclave memory; enclave memory cannot be accessed by non-enclave code; and no enclave can be accessed after it has been killed. These guarantees ensure that an attacker who cannot access enclave memory cannot access any high-security values.

Finally, the type system enforces that all escape hatches compute using only values from memory locations that are immutable (read-only). Equivalently, no escape hatch can access mutable memory or register values. This ensures that the information used to compute the escape hatch is contained only the initial memory, regardless of where the program is in its execution.

Because expressions do not output values, and change neither the variable context nor kill set, the typing judgment for expressions is much simpler and is not parametrized by a program counter, variable context, or set of unset conditions.

### 2.4 SIMPE: Simplified IMPE

SIMPE is a simplified model of IMPE for which we verify security properties. SIMPE makes the following simplifications:

1. SIMPE does not support erasure security policies. As a result, SIMPE also has no notion of conditions, which are used for erasure policies in IMPE. Furthermore, there is no notion of level \( \top \) in SIMPE because \( \top \) is only meaningful in the context of erasure policies.

2. SIMPE assumes that all enclaves are always alive, whereas IMPE has a \( \text{kill}\) command for killing an enclave during execution.

3. SIMPE adds the restriction that the initial memory \( m_{\text{init}} \) has no locations in it, and that no program can update memory to contain a location. This eliminates the possibility of nested pointers.

(1) and (2) allow us to simplify IMPE configurations by removing kill sets, and to simplify the IMPE typing judgment by removing the set of unset conditions.

---

**Figure 3:** IMPE expression and command typing judgments

**Figure 4:** The typing judgment for a \( \text{Cwhile\ e\ c}\) command that prevents leaks through implicit flows.

**Figure 5:** A program with implicit flows that is not well-typed when \( \Sigma(\text{password}) = T\text{nat}_H \) and \( \Sigma(\text{guess}) = T\text{nat}_H \).
3. SIMPE Noninterference Security

We prove any well-typed SIMPE program to be secure under two attacker models: (1) a passive attacker that observes the program’s execution and (2) an active attacker that can modify non-enclave code. Both attackers observe the program’s entire execution only on the low output channel \(L\).

An attacker’s knowledge is intuitively defined as the locations in memory for which the attacker knows something about the value there. Thus, if an attacker has more knowledge, she more precisely knows the contents of the initial memory.

An attacker’s knowledge derives from the entropy set of an attacker’s knowledge, which is the set of memories that the attacker believes could generate the same output that she observed. The fewer memories the attacker believes are possible (the lower her entropy), the more precise her knowledge.

A program is secure if an attacker knows at most about the contents of the initial memory at locations that (1) have a low security policy, or (2) are used to compute escape hatches. Equivalently, a program is secure if the attacker’s entropy set contains at least every memory that:

(1) contains the same values as the initial memory at locations with a low security policy, and

(2) contains values in locations such that all escape hatches in the program evaluate to the same value as when evaluated on the initial memory.

3.1 Overview of Security Verification Strategy

Our goal is to prove Theorem 1 (presented in Section 3.7), which states that if a SIMPE program is well-typed, then the SIMPE program is secure under the passive and active attacker models. Our proof follows the proof technique of Pottier and Simonet [10], which Gollamudi and Chong use in their pen-and-paper security proof. The Pottier and Simonet technique aids in reasoning about noninterference and sensitive information flows in a program because it turns reasoning about two executions into reasoning about one.

At a high level, the strategy simultaneously executes a well-typed SIMPE program on two different initial memories using a new calculus SIMPE\(^2\), which we describe in the next section. The first memory, \(m_{\text{left}}\), is the initial memory upon which the attacker observes the SIMPE program execution. The second memory \(m_{\text{right}}\) is any memory that must be in the attacker’s entropy set in order for the program to be secure.

By tracking certain invariants of the SIMPE\(^2\) execution, we prove that the execution of any well-typed SIMPE program on \(m_{\text{left}}\) produces the same observable output as the program’s execution on \(m_{\text{right}}\). This ensures that the attacker must believe memory \(m_{\text{right}}\) is a possible initial memory, so the SIMPE program is secure by definition.

3.2 SIMPE\(^2\)

We model a new calculus, SIMPE\(^2\), which executes the same SIMPE command on two different initial memories \(m_{\text{left}}\) and \(m_{\text{right}}\) simultaneously, where \(m_{\text{left}}\) and \(m_{\text{right}}\) are the memories described above. SIMPE\(^2\) tracks two sets of SIMPE configurations in the form of pairs of SIMPE traces, registers, and memories. SIMPE\(^2\) supports pairs of values (VPairs) as a value type for when the two executions evaluate to different values.

For clarity, we refer to the register, memory, and trace of the first execution as \(r_{\text{left}}, m_{\text{left}},\) and \(t_{\text{left}}\), and the register and memory of the second execution as \(r_{\text{right}}, m_{\text{right}},\) and \(t_{\text{right}}\). We merge \(r_{\text{left}}\) and \(r_{\text{right}}\) to get a reg2, and can project a reg2 value to get back the values of \(r_{\text{left}}\) and \(r_{\text{right}}\) (and similarly for memory and traces). SIMPE\(^2\) semantic judgments are simple extensions of SIMPE semantics. Figure 6 shows the large-step judgment and definitions of SIMPE\(^2\) syntax.

We include inference rules that allow SIMPE\(^2\) to model how a pair of SIMPE executions may perform...
**Definition** command_secure (c: com) : Prop :=

let mem_init2 = merge(mleft, mright) in

com_step2 None (c, reg_init2, mem_init2)
(r', m') t2
→ (∀ loc, SG(loc) = σt ⇒ mleft(loc) = mright(loc))
→ (∀ e ∈ EscapeHatches(c),
    exp_step md (e, r, mleft) v ̸⇒
    exp_step md (e, r, mright) v)
→ observations_low_channel (project t2 left) =
    observations_low_channel (project t2 right).

Figure 7: Secure program definition using SIMPE².

**Definition** config2_wt : mode → policy → var_context →
com_config2 → var_context → Prop :=

com_wt md pc Γ c Γ'
∧ (∀ x, r(x) = VPair v1 v2 ∧ Γ(x) = σd
  → sl = H ∧ md ̸= None)
∧ (∀ loc, m(loc) = VPair v1 v2 ∧ SG(loc) = σd
  → sl = H ∧ δ(loc) ̸= None).

finalconfig2_wt : mode → policy →
var_context → final_config2 → Prop.

Figure 8: Typing judgments for SIMPE² configurations and final configurations.

(a) The command in the configuration c is well-typed.
(b) If r_left and r_right contain different values for the same variable, then that variable is typed with a high security policy and the current mode of execution is in an enclave.
(c) If m_left and m_right contain different values at the same location, then that location is typed with a high security policy and the memory location belongs to an enclave.

A final configuration is well-typed if invariants (b) and (c) hold. The signature for the corresponding judgment is shown in Figure 8.

3.3 SIMPE² Adequacy

We first prove the adequacy of SIMPE²; SIMPE² is both sound and complete. This allows us to extend our reasoning about the pairs of execution traces and values produced by a SIMPE² execution to execution traces and values produced by two SIMPE executions.

**Lemma 1. (Soundness)**

com_step2 md (c, r, m) (r', m') t ̸⇒
com_step md (c, project r left, project m left)
(project r' left, project m' left)
(project t left)
∧
com_step md (c, project r right, project m right)
(project r' right, project m' right)
(project t right)
∧

Soundness ensures that any SIMPE² large-step execution corresponds to two individual SIMPE large-step executions. The proof of soundness follows by straightforward induction on the SIMPE² large-step derivation.

**Lemma 2. (Completeness)**

com_step md (c, rleft, mleft)
(rleft', mleft') tleft ̸⇒
com_step md (c, rright, mright)

We need to show that SIMPE² configurations always maintain certain invariants in order to make claims about how and when observable differences between the two SIMPE registers and memories tracked by an SIMPE² execution occur. We use a new configuration typing judgment config2_wt to capture these invariants; the definition is shown in Figure 8. A configuration is well-typed (config2_wt) when:

3.4 SIMPE² Well-Typed Configuration Invariants
Intuitively, invariants (b) and (c) ensure that all values that differ between \( r_{left} \) and \( r_{left} \), and \( m_{left} \) and \( m_{right} \) are typed as high-security values and accessed only in an enclave. This will allow us to prove that an attacker who cannot modify enclave code cannot alter the code in any way to cause SIMPE\(^2\) to produce a pair of traces with different values.

### 3.5 Configuration Type Preservation

We prove two lemmas about configuration type preservation. Given a well-typed initial configuration, Lemma 3 asserts that the final configuration is well-typed, and Lemma 4 asserts that intermediate configurations of the execution are well-typed. Lemma 4 also ensures that intermediate configurations are executed with a program counter that has security at least as high as the initial configuration.

**Lemma 3. (Final Configuration Invariant Preservation)**

\[
\text{config2} \cdot \text{wt md pc } \frac{\Gamma \Gamma'}{c \text{config2} \Gamma' \rightarrow} \\
\text{com} \cdot \text{step2 m} \frac{\text{cconfig2 finalconfig2} \rightarrow \text{t2 }}{t2} \\
\frac{\text{finalconfig2} \cdot \text{wt md pc } \frac{\Gamma}{\Gamma} \text{finalconfig2}}{.}
\]

**Lemma 4. (Intermediate Configuration Invariant Preservation)**

\[
\text{config2} \cdot \text{wt md pc } \frac{\Gamma \Gamma'}{c \text{config2} \Gamma' \rightarrow} \\
\text{com} \cdot \text{step2 m} \frac{\text{cconfig2 finalconfig2} \rightarrow \text{t2 }}{t2} \\
\frac{\text{intermediate_config (cconfig2) (cconfig2)} \rightarrow \text{com} \cdot \text{step2 m} \frac{\text{cconfig2_t2} (\text{finalconfig2}) (\text{t2 }) \rightarrow \text{t2}}{t2}}{t2} \\
\frac{\text{config2} \cdot \text{wt md pc } \frac{\text{config2} \cdot \text{im} \frac{\text{config2} \cdot \text{im}}{\text{config2} \cdot \text{im}} \frac{\text{config2} \cdot \text{im} \frac{\text{config2} \cdot \text{im}}{\text{config2} \cdot \text{im}} \frac{\text{config2} \cdot \text{im}}{\text{config2} \cdot \text{im}} \Gamma \Gamma'}{\text{config2} \cdot \text{im}}} \rightarrow \text{pc} \cdot \text{im} \geq \text{pc}.
\]

The proof of both Lemma 3 and Lemma 4 proceeds by induction on the large-step semantics \((\text{com} \cdot \text{step2})\). Adequacy of SIMPE\(^2\) is necessary to extend typing judgments from SIMPE to SIMPE\(^2\) and assert statements about a well-typed command executing in the SIMPE\(^2\) calculus.

The key observation used to prove these two preservation lemmas is Lemma 5, which states that if an expression \( e \) evaluates to a \( \text{VPair} \), \( e \) is typed with high security and executes in an enclave.

**Lemma 5. (Expression Pair Invariants)**

\[
\text{exp} \cdot \text{step2 m} \frac{\text{e r m} \frac{\text{VPair v1 v2}}{\text{val} \cdot \text{wt} \frac{\text{e } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{sl } \frac{\text{H } \text{md }}{\neq \text{None}}} {\text{H}}}}{\text{H}} \\
\text{sl } = \text{H } \text{md} \neq \text{None}.
\]

Proving this helper lemma requires that we extend expression types to value types. To do so, we prove the additional lemma of value type preservation, where \( \text{val} \cdot \text{wt} \) is defined similarly to \( \text{exp} \cdot \text{wt} \).

**Lemma 6. (Value Type Preservation)**

\[
\text{exp} \cdot \text{step2 m} \frac{\text{e r m} \frac{\text{VPair v1 v2}}{\text{val} \cdot \text{wt} \frac{\text{e } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow}} {\text{v}}} {\text{v}}} \\
\text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow}} {\text{v}} \\
\text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow}} {\text{v}} \\
\text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow}} {\text{v}} \\
\text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow \text{val} \cdot \text{wt} \frac{\text{v } \frac{\text{sigma} \frac{\text{md }}}{\text{H}} \rightarrow}} {\text{v}}
\]

Lemma 5 allows us to verify that an assignment of a \( \text{VPair} \) value to a register, or an update of a memory location with a \( \text{VPair} \) value, only occurs when the value has a security level \( H \), and that the update or assignment happens in an enclave. We consider two cases:

* \( e \) in \( \text{update(loc,e)} \) or \( \text{assign(x,e)} \), which assigns \( x \) the result of evaluating escape hatch \( e \). Thus any assignment or update performed by the subcommand must assign or update in enclave mode with values of high security.

The next step is to prove that only assignments and updates introduce \( \text{VPair} \) values into registers or memory. The only other way that a register or memory location can be modified is via the \( \text{declassify(x,e)} \) command, which assigns \( x \) the result of evaluating escape hatch \( e \). We prove that, by definition, an escape hatch can never evaluate to a \( \text{VPair} \), and therefore a \( \text{declassify} \) command cannot introduce a \( \text{VPair} \) into a register.

Because registers and memories gain \( \text{VPair} \) values only through an assignment or an update, Lemma 5 allows us to prove that invariants (b) and (c) (from Section 3.4) hold throughout the execution of any well-typed configuration. The type system guarantees that, throughout a command’s execution, invariant (a) holds.

### 3.6 Observational Equivalence

Using the two lemmas about invariant preservation, we prove observational equivalence. Observational equivalence states that a well-typed SIMPE program produces equivalent output traces on the low channel when executing on \( m_{left} \) and \( m_{right} \), where \( m_{left} \) is the initial memory upon which the attacker observes the SIMPE
program execution, and \( m_{\text{right}} \) is any memory that must be in the attacker’s entropy set.

**Lemma 7. (Observational Equivalence)**

\[
\text{config2_wt} \; \text{md} \; \text{pc} \; \Gamma \; \text{c} \; \text{reg_init2}, \; \text{mem_init2} \; \Gamma' \rightarrow \\
\text{com_step2} \; \text{md} \; \text{c} \; \text{config2} \; \text{final_config2} \; \text{t2} \rightarrow \\
\text{observations_low_channel} \; (\text{project_trace} \; \text{t2} \; \text{left}) = \\
\text{observations_low_channel} \; (\text{project_trace} \; \text{t} \; \text{right}).
\]

The proof of observational equivalence proceeds by induction on the large-step semantics (\text{com_step2}).

The base case is the output(e, sl) command, which produces the only observable effects of the program’s execution. If the program never outputs a VPair to the low channel, then the observable effects of the program executing on \( m_{\text{left}} \) are equivalent to the observable effects of the program executing on \( m_{\text{right}} \). The preservation lemmas state that throughout execution, any VPair in a register or memory location must be high-security and accessible only when in an enclave. Because e is well-typed (which follows from config2_wt), we know that if \( e \) evaluates to a VPair, \( e \) has a high-security type and \( s_l \neq L \). Otherwise, the value output to \( s_l \) is not a VPair, and the attacker observes either no value if \( s_l = H \) or the same value.

The other cases of interest are those in which an expression \( e \) causes a dividing execution. From Lemma 5, we know \( e \) must be typed with high-security. This implies that the conditional branch commands, or the called function command, must be well-typed with a high-security program counter. Lemma 8 asserts that a well-typed command with a high-security program counter cannot output any values to a low-security channel. This allows us to prove that when the execution of \( \text{SIMPE}^2 \) diverges on \( m_{\text{left}} \) compared to \( m_{\text{right}} \), the execution produces no observable differences.

**Lemma 8. (Secure program counter execution produces no observable output)**

\[
\text{com_wt} \; \text{md} \; \text{pc} \; \Gamma \; \text{c} \; \Gamma' \land \text{pc} = H \rightarrow \\
\text{com_step2} \; \text{md} \; (\text{c}, \text{r}, \text{m}) \; (\text{r}', \text{m}') \; \text{t2} \rightarrow \\
\text{observations_low_channel} \; \text{t2} = [\mathcal{L}].
\]

### 3.7 Noninterference Security

From observational equivalence, our final security theorem follows: a well-typed \( \text{SIMPE}^2 \) program is secure in the presence of both a passive attacker who either simply observes the program’s execution, and an active attacker who can modify non-enclave code.

**Theorem 1. (Well-typed Programs are Secure)**

If \( c \) is well-typed, then the configuration \( (c, \text{reg_init2}, \text{mem_init2}), \text{where reg_init2} \) is a pair of registers initialized to all 0 values, and \( \text{mem_init2} = \text{merge}(m_{\text{left}}, m_{\text{right}}) \) is well-typed:

\[
\text{config2_wt} \; \text{md} \; \text{pc} \; \Gamma \; \text{c} \; \Gamma' \rightarrow \text{command_secure} \; \text{c}.
\]

**Lemma 7** asserts that the execution on \( m_{\text{left}} \) produces the same observable output as the execution on \( m_{\text{right}} \). We therefore conclude that \( m_{\text{right}} \) is in the attacker’s entropy set. Thus, a well-typed \( \text{SIMPE} \) command is secure for a passive attacker.

We next extend our reasoning to the active attacker model, in which the attacker can modify non-enclave code. We need to show that an active attacker’s edited program cannot produce different observable outputs when it executes on \( m_{\text{left}} \) than when it executes on \( m_{\text{right}} \). An active attacker can read, modify, and output any non-enclave memory without regard for the security policy on the memory location, or the security policy on the value at that memory location. An active attacker can also output any register’s value when not executing code in an enclave. Our proof of config2_wt preserves ensures if execution on \( m_{\text{left}} \) ever contains a value in a register or in memory that differs from that of the execution on \( m_{\text{right}} \), then the execution is taking place in an enclave. This ensures that an active attacker cannot output any register or memory value that would be different in an execution on \( m_{\text{left}} \) from an execution on \( m_{\text{right}} \), because an active attacker cannot access memory in an enclave, nor modify enclave code.

### 4. Security Verification Experience

When specifying the \( \text{IMPE} \) and \( \text{SIMPE} \) calculi, we make slight modifications to Gollamudi and Chong’s original \( \text{IMPE} \) model to aid verification. These modifications follow a common theme of making implicit assumptions explicit.

We make explicit that the location context \( \Sigma \) is constant through execution. We introduce \( \Sigma \) as a global parameter to the model, rather than threading the context through the program’s typing judgment. By explicitly specifying that \( \Sigma \) does not change, we are able to eliminate concepts from the original model, such as the security policy on the memory location, or the security policy on the value at that memory location. An active attacker can also output any register’s value when not executing code in an enclave. Our proof of config2_wt preserves ensures if execution on \( m_{\text{left}} \) ever contains a value in a register or in memory that differs from that of the execution on \( m_{\text{right}} \), then the execution is taking place in an enclave. This ensures that an active attacker cannot output any register or memory value that would be different in an execution on \( m_{\text{left}} \) from an execution on \( m_{\text{right}} \), because an active attacker cannot access memory in an enclave, nor modify enclave code.

\[\text{Gollamudi and Chong have a single typing context for both variables and locations. We separate this context into two contexts, } \Gamma \text{ and } \Sigma, \text{ to better model that the location context is fixed and the variable context changes during execution.}\]
rity specification (which maps memory locations to security policies) and to simplify proofs. For example, we do not need to assume that Σ and this security specification are consistent at every step. In addition, we do not need to prove the preservation of well-typed location contexts (which states that all locations at high security belong in an enclave); we assume that Σ is well-typed before the execution begins, and therefore it remains well-typed through the execution.

We also instrument the command execution to produce ghost output events in the trace whenever an update to a memory location, or assignment to a register, has occurred, and make explicit any updates or assignments performed during execution. This allows us to determine exactly when a register or memory location changes value.

4.1 Simplifications from IMPE to SIMPE

We verify the security of a simplified version of the IMPE calculus, which we call SIMPE. Removing erasure policies allows us to simplify reasoning about the current security level of each value. Our proofs track only a static security level for each value, which eliminates the need to thread the set of unset conditions throughout the proofs. Removing erasure policies (and security level ⊤) also allows us to ignore killed enclaves: killed enclaves allow us to have values typed with security ⊤, but do not provide other relevant semantics for the sake of our security proof.

We also simplify the attacker model by requiring that the attacker observe the program’s execution from the start of execution; Gollamudi and Chong allow the attacker to begin observing the execution after the program has already executed partway. Our simplification allows us to make statements about the entire trace produced by the program’s execution, rather than an arbitrary portion of the trace.

4.2 Axioms

Here we describe the parts of the SIMPE and SIMPE\(^2\) model that we specified using axioms, and our reasons for doing so.

The first is the subsumption typing rule, which allows a command well-typed at high security to be well-typed at a lower security level. Because subsumption is an axiom instead of a typing rule, each distinct command has only one typing judgment that makes it well-typed, and an inversion on \texttt{com_wt} produces only one possible constructor.

**Axiom 1. (Subsumption Typing Rule)**

\[
\begin{align*}
\text{com_wt } &\quad \text{md } pc1 \ \Gamma_1 \ c \ \Gamma_1' \rightarrow \\
\text{pc2 } &\quad \leq \ pc1 \ \wedge \ \Gamma_2 \leq \ \Gamma_1 \ \wedge \ \Gamma_1' \leq \ \Gamma_2' \rightarrow \\
\text{com_wt } &\quad \text{md } pc2 \ \Gamma_2 \ c \ \Gamma_2'.
\end{align*}
\]

We also include an axiom that there are no pointers in memory, which allows us to eliminate the possibility of nested pointers. This allows us to define a well-typed escape hatch. One of the key properties of an escape hatch is that all values in the escape hatch derive from immutable locations. If the initial memory contains nested pointers, a value in an immutable location used by the escape hatch computation may point to a mutable location, and cause the escape hatch to evaluate to different values at different points of the execution. Gollamudi and Chong assume the existence of a \texttt{AllLocImmutable(e)} function, but do not clarify how to compute the “locations in e.” To avoid checking for all reachable locations in an escape hatch, which requires recursing on nested location types, we specify that the memory of an SIMPE program contains no pointers.

**Axiom 2. (No Pointers)**

\[
\forall \ loc, m(loc) \neq \emptyset \ loc \ loc'
\]

Axioms 3 and 4 assert properties about the initial program state. Axiom 3 ensures that all values in the initial memory are well-typed, and that their types correspond to the types fixed by the location context Σ. We require this to make any assertion about the well-typeness of values derived from a dereference of memory during execution. Gollamudi and Chong do not state this guarantee about initial state, which we found to be essential to proving any properties about well-typed expressions and values.

**Axiom 3. (Initial Memory Well-Formed)**

\[
\forall \ loc, \ match \ \text{minit } l \ \text{with} \\
\quad \mid \ V\lambda \ c \Rightarrow \Sigma(l) = T\lambda \text{ndef } \wedge \text{com_wt } \text{md } pc \ \Gamma_m \ c \ \Gamma_p \\
\quad \mid \ Vloc \ l \Rightarrow \text{False} \\
\quad \mid \ Vnat \ n \Rightarrow \Sigma(l) = T\text{nat}\text{ndef} \\
\end{align*}
\]

Axiom 4 connects all register and memory state produced during the program execution to the initial register and memory state: all registers and memories can be derived by executing some command starting from the initial state. This allows us to make claims about the execution’s history.
Axiom 4. (Connection to Initial State)
\[
\forall \text{reg}, \forall \text{mem}, \\
\text{com}\_\text{step None (c, reg\_init, mem\_init) (reg, mem) tr.}
\]

Axiom 5 states that if a register ever contains either a function or location value, then there must have been a well-typed expression \( \lambda \) or \( \text{loc} \) that evaluated to that value (and this expression must have occurred within an assignment command). Similarly, if a function is in memory, then the function already was contained in the initial memory, or there was a well-typed expression that evaluated to that value.

Axiom 5. (Presence of Values in Registers and Memory)
\[
\begin{aligned}
\text{reg}(x) &= \lambda\text{c} \land \Gamma(x) = \lambda\text{c} \\
&\iff \text{exp}\_\text{type md} \Gamma(\lambda\text{c}) \lambda\text{c} \\
\text{mem}(\text{loc}) &= \lambda\text{c} \land \Gamma(\text{loc}) = \lambda\text{c} \\
&\iff \text{exp}\_\text{type md} \Gamma(\lambda\text{c}) \lambda\text{c} \\
&\lor (\text{mem\_init}(\text{loc}) = \lambda\text{c})
\end{aligned}
\]

Axiom 5 follows from Axiom 4. We know that any function or location value comes from one of three sources: (1) an expression \( \lambda \) or \( \text{loc} \), (2) another register, or (3) a memory location. We know that the initial registers start out with value 0, and the initial memory is \( \text{mem\_init} \) which is well-formed. In cases (2) and (3), we can consider how the value was placed in the register or memory location. Again, we have that the value placed into the register or memory must have come from one of the three sources. Axiom 4 guarantees that eventually this reasoning will terminate, because we will either reach the initial state of the program or find that the value comes from a basic expression \( \lambda \) or \( \text{loc} \).

We present Axiom 5 as an axiom instead of a lemma because the reasoning described above requires instrumenting our semantic judgments to record the history of the execution. For example, adding a counter to \( \text{com}\_\text{step} \) will allow easy backward reasoning (each step in the execution increments the counter, and when the counter is 0, the execution is in its initial state). However, this requires reworking our entire model and proof infrastructure.

The connection between register and memory state at an arbitrary point in a command’s execution, and the initial \( \text{reg\_init} \) and \( \text{mem\_init} \) state, is implicitly assumed throughout the proof presented by Gollamudi and Chong. We attribute our difficulty in proving Axiom 5 to the fact that this assumption must be made explicit in Coq, which we realized too late in the proof process.

The same logic used to justify Axiom 5 justifies the only axiom we make in modeling SIMPE^2:

Axiom 6. (Well-Formed VPairs)
\[
\begin{aligned}
v &= \text{VPair} v_1 v_2 \iff v_1 \neq v_2 \\
&\lor ((v_1 = \text{Vnat} n_1 \land v_2 = \text{Vnat} n_2) \lor \{v_1 = \lambda\text{c1} \land v_2 = \lambda\text{c2}\})
\end{aligned}
\]

This axiom claims that no pair is a pair of the same value, and that pairs always contain values of the same type. To prove this axiom, we need to step back through the derivation to the initial state, and prove that only well-formed pairs are produced by an SIMPE^2 execution.

4.3 Evaluating the Pen-and-Paper Proof
During the verification process, we discovered a few incorrectly stated assumptions in Gollamudi and Chong’s pen-and-paper proof of IMPE security. Gollamudi and Chong have conflicting specifications of the subtyping relation between contexts (i.e., \( \Gamma_1 \leq \Gamma_2 \)). We are able to prove Lemma 3 using this definition of the subtyping relation:

\[
\forall x, \Gamma_1(x) = \sigma_{s1} \land \Gamma_2(x) = \sigma_{s2} \rightarrow s1 \leq s2
\]

Gollamudi and Chong require that the domains of \( \Gamma_1 \) and \( \Gamma_2 \) be equivalent, whereas we require only that the domain of \( \Gamma_2 \) be \textit{at least as large} as that of \( \Gamma_1 \). This change is necessary to use the subsumption rule in the proof of Lemma 3; it also appears that Gollamudi and Chong use this relaxed definition in their presentation of the proof their corresponding lemma.

Gollamudi and Chong also assume Lemma 5 in their proofs of security against the passive attacker model, whereas we provide an explicit proof of this lemma.

We also found several mistakes and typos in the pen-and-paper proof. One of the more significant was an incorrect statement of the \textit{while-div} rule for SIMPE^2. Gollamudi and Chong state that a \textit{while} e do c command is equivalent to executing either a \textit{skip} command or the command c. This does not allow a \textit{while} command to execute c more than once. We discovered this incorrect specification while attempting to prove the adequacy of SIMPE^2.
Other errors included using incorrect variable names in a judgment, and copy-paste errors from incorrectly copying a very similar proof from one inductive case to another. Given the number of inductive cases in each proof, we found it rewarding to do the proof in Coq, where simple errors are caught automatically. However, as shown by our verification, the general intuition behind the pen-and-paper proof was correct.

In general, we found that while it is sufficient for a pen-and-paper proof to assert that a lemma holds “by straightforward induction,” proving such lemmas in Coq may require potentially hours of work. Furthermore, intuitive reasoning about simple concepts, such as how assignments affect registers, are written in one sentence on paper, but require hundreds of lines in Coq to prove.

4.4 Evaluating the SIMPE\(^2\) Proof Technique

We also evaluate how the Pottier and Simonet technique for proving noninterference translates to a Coq setting. Modeling SIMPE\(^2\) essentially amounts to copy-pasting the SIMPE model with minor changes, but is ultimately useful because we induct on a single \texttt{com\_step2} instead of performing dual induction on two \texttt{com\_steps}.

The drawback of the SIMPE\(^2\) technique is that an entirely new SIMPE\(^2\) must be created, and the same adequacy and preservation lemmas proved, if SIMPE’s typing judgments or semantics are even slightly modified. This technique does not lend itself well to modular changes or extensions. Because our Coq proofs rely on induction and inversions on semantic and typing judgments, it would be difficult to design proofs that could be efficiently reused for any new SIMPE\(^2\) model.

5. Automatic Enclave Placement

The placement of code and data into enclaves is a tedious process that can be effectively automated. We model a constraint-based translation scheme developed by Gollamudi and Chong. This scheme guarantees that any translation that meets the scheme’s constraints will produce a well-typed IMPE program given a well-typed input program written in a security-typed, but enclave-agnostic calculus IMPS. Since IMPE programs are secure against certain low-level attackers, this means that a programmer can write code in a relatively high-level calculus and get low-level security guarantees.

Fundamentally, the constraints work by forcing the translation to place sensitive code and data in enclaves. What makes the scheme more interesting, and more interesting to verify, is that it gives the translation flexibility about how to manage the enclaves, both in terms of how to distribute code and data between different enclaves, and when to kill enclaves. For example, one translation might place all code and data in a single enclave to reduce the performance cost incurred by entering and exiting enclaves, while another translation might minimize the size and lifespan of enclaves to reduce the possibility of an adversary accessing sensitive information through a vulnerability in enclave code.

In the following subsections we discuss IMPS, the translation scheme, and how we verified the correctness of the scheme. For the purposes of presentation, we simplify the model of IMPS, the translation scheme, and IMPE in this discussion. In particular, we omit the program counter security policy, the set of unset conditions (used for enforcing erasure policies), and typing contexts from the judgments for the translation and the two calculi, as they are irrelevant to what makes the scheme interesting and clutter the notation. Determined readers can refer to our Coq implementation for the full, unrefined model.

5.1 Modeling IMPS

The IMPS calculus is very similar to IMPE, except that it has no notion of enclaves.\(^3\) As a result, it does not have modes, kill sets, or enclave or kill commands. The IMPS syntax is given in Figure 11. Unlike in IMPE, a sequence of commands is not itself considered a command. Instead, it is considered a \texttt{seq}, a distinct syntactic category. This guarantees the absence of nested sequences of commands. Additionally, every “subcommand” of an IMPS expression or command is a \texttt{seq}, instead of an arbitrary command, and a top-level IMPS program is also a \texttt{seq}. This syntactic structure is important because the translation rule for sequences of commands determines the placement and destruction of enclaves and thus drives the entire translation.

Typing judgments for IMPS expressions and commands are modeled by the following inductive types:

\[
\text{exp\_wt : exp \rightarrow type \rightarrow deriv \rightarrow Prop}
\]

\(^3\)Where there is potential ambiguity, we distinguish between IMPS and IMPE material in our pseudocode by treating the former as coming from module \texttt{S} and the latter from module \texttt{E} (e.g., \texttt{S.exp\_wt} versus \texttt{E.exp\_wt}).
The typing judgment seq_wt has the same signature as com_wt. The judgments are effectively simplified versions of the IMPE typing judgments, except that we add a new term of type deriv. This term captures information about the derivation of the typing judgment that is otherwise lost. For instance, consider the definition of the IMPS typing judgment for the output command:

\[
\text{exp_wt} \rightarrow \text{com_wt} \rightarrow \text{Prop}
\]

Most of the constraints for translating expressions and commands are straightforward. Consider, for instance, the translation scheme judgment for the output command:

\[
\text{exp_trans} \text{e} \text{t} \text{drv} \text{md} \delta e' t' \wedge \text{In} \text{md} K \rightarrow \text{com_trans} (\text{S.Coutput} \text{e} \text{chan}) (\text{Deriv} \text{t} \text{drv}) \text{md} \delta \text{K} (\text{E.Coutput} \text{e' chan}) \text{K}
\]

This judgment (1) constrains e' to be the translation of e, (2) constrains the set of killed enclaves K before the command E.Coutput e' chan to not include the mode md that the command will run under, and (3) constrains the set of killed enclaves after the command to be the same as the set before the command.

The heart of the translation scheme lies in the definition of seq_trans, which has a single constructor given in Figure 9. The translation constraints take the form of constraints between members of various lists: the list of IMPS commands (ecoms), a list of IMPE commands (coms), a list of modes (mds), and three lists of kill sets (K1, K2, and K3). For relevant i, the first premise of the constructor makes the following requirements:

1. The ith member of scoms translates to the ith member of ecoms, which runs in mode mdsi with initial kill set K1_i and finishes with kill set K2_i.
2. K1_{i+1} is the union of K2_i and K3_i. Since K2_i is the kill set immediately after ecomsi finishes and K1_{i+1} is the kill set immediately before ecoms_{i+1}, this premise allows the translation to insert kill commands between ecoms_i and ecoms_{i+1} that destroy the enclaves in K3_i, a set which is underspecified (although subject to restrictions given in the following premises).
3. K2_i and K3_i are disjoint.
4. If the sequence runs in an enclave, then each subcommand must run in the same enclave.
5. If two consecutive commands in ecoms run in the same enclave, then no kill commands can be inserted between them.

The second premise of the constructor involves process_seq, which is a function that consumes a list of IMPE commands ecoms, an overall mode md, a list of modes mds, a list of kill sets K3s, and produces a new sequence of commands ecoms' in which enclave and kill commands have been placed appropriately. A key decision was to model this function as an inductive type that also ranges over the extra parameters K1s, K2s, and Ksout, which are all lists of kill sets. Ksout allows...
us to track the kill sets before and after each command in ecoms', and therefore more easily prove that each command is each well-typed.

A proposition in the form

\[
\text{process_seq ecoms } \text{md } \text{mds } K1s \hspace{1pt} K2s \hspace{1pt} K3s \hspace{1pt} Ksout \rightarrow
\text{seq_trans } (S.Cseq \hspace{1pt} scoms) \hspace{1pt} (Derivseq \hspace{1pt} drvs) \hspace{1pt} \text{md } \delta \hspace{1pt} (h \hspace{1pt} Ksout) \hspace{1pt} (E.Cseq \hspace{1pt} ecoms') \hspace{1pt} (last \hspace{1pt} Ksout)
\]

The proof precedes by mutual induction on the seq_trans, com_trans, and exp_trans judgments. Most of the cases are straightforward; we focus on the seq_trans case.

In this case, we need to prove that the command E.Cseq ecoms' is well-typed when the first member of Ksout is the set of killed enclaves before E.Cseq ecoms' runs and the last member of Ksout is the set of killed enclaves after the command runs. Since ecoms' and Ksout are artifacts of the process_seq judgment, it is helpful to frame this goal as a helper lemma specifically about process_seq:

**LEMMA 9. (Process Sequence is well-typed)**

\[
(\forall 0 \leq i < \text{length } \text{ecom}s, \hspace{1pt} \text{E.com_wt } \hspace{1pt} \delta \hspace{1pt} K1s[i] \hspace{1pt} \text{ecom}s[i] \hspace{1pt} K2s[i]) \rightarrow
... \hspace{1pt} (* \hspace{1pt} \text{premises from seq_trans } *) ... \hspace{1pt} \text{process_seq ecoms } \hspace{1pt} \text{md } \hspace{1pt} \text{mds } K1s \hspace{1pt} K2s \hspace{1pt} K3s \hspace{1pt} \text{ecom}s' \hspace{1pt} \text{Ksout } \rightarrow \hspace{1pt} \text{E.com_wt } \hspace{1pt} \delta \hspace{1pt} (h \hspace{1pt} \text{Ksout}) \hspace{1pt} (E.Cseq \hspace{1pt} \text{ecom}s') \hspace{1pt} (last \hspace{1pt} \text{Ksout}).
\]

The first premise corresponds to the induction hypothesis generated during the seq_trans case of the proof of Theorem 2 (for brevity we omit many premises that come from the seq_trans constructor).

We use two lemmas to prove this helper:

**LEMMA 10. (Process Kill is well-typed)**

\[
K2 \cap K3 = \emptyset \hspace{1pt} \land \hspace{1pt} \text{process_kill } K2 \hspace{1pt} K3 \hspace{1pt} \text{kcoms } \text{Ksout } \rightarrow \hspace{1pt} \forall 0 \leq i < \text{length } \text{kcom}s, \hspace{1pt} \text{E.com_wt } \hspace{1pt} \delta \hspace{1pt} \text{Ksout[i]} \hspace{1pt} \text{kcoms[i]} \hspace{1pt} \text{Ksout[i+1]}.
\]

**LEMMA 11. (Process Sequence is well-typed 2)**

\[
(\forall 0 \leq i < \text{length } \text{ecom}s, \hspace{1pt} \text{E.com_wt } \hspace{1pt} \delta \hspace{1pt} K1s[i] \hspace{1pt} \text{ecom}s[i] \hspace{1pt} K2s[i]) \rightarrow
... \hspace{1pt} (* \hspace{1pt} \text{premises from seq_trans } *) ... \hspace{1pt} \text{process_seq ecoms } \hspace{1pt} \text{md } \hspace{1pt} \text{mds } K1s \hspace{1pt} K2s \hspace{1pt} K3s \hspace{1pt} \text{ecom}s' \hspace{1pt} \text{Ksout } \rightarrow \hspace{1pt} \forall 0 \leq i < \text{length } \text{ecom}s, \hspace{1pt} \text{E.com_wt } \hspace{1pt} \delta \hspace{1pt} \text{Ksout[i]} \hspace{1pt} \text{ecom}s'[i] \hspace{1pt} \text{Ksout[i+1]}.
\]
The \(K_{\text{out}}\) term is key to both these lemma statements. The claim we are making is that, in both the case of \(\text{process}_\text{kill}\) and \(\text{process}_\text{seq}\), the \(i\)th command in the output sequence is well-typed when preceded by the \((i+1)\)th kill set of \(K_{\text{out}}\) and followed by the \(i\)th kill set of \(K_{\text{out}}\). While the definitions of \(\text{process}_\text{seq}\) and \(\text{process}_\text{kill}\) do not require such a \(K_{\text{out}}\) term to specify the required functionality, we included it in the judgments as a ghost variable to make the correctness property easier to state and prove.

Proving Lemma 10 amounts to proving that a sequence of kill commands is well-typed, and is straightforward given the hypothesis that \(K_2\) and \(K_3\) are disjoint and the definition of \(\text{process}_\text{kill}\). The proof for Lemma 11 uses Lemma 10 as a helper lemma and is substantially more involved. The proof proceeds by induction on the \(\text{process}_\text{seq}\) judgment and involves showing that an arbitrary command of the output sequence is well-typed in each of the three cases corresponding to the three constructors. Since the command might be one of the original commands, a new enclave command that houses a subsequence of the original commands, a new kill command, or the output of the recursive call to \(\text{process}_\text{seq}\) on a suffix of the input sequence of commands, there are many cases to cover in the proof. Nonetheless, the proof is conceptually straightforward overall.

6. Related Work

6.1 Verification of enclave security

Moat [13] uses BoogiePL [4] to verify confidentiality properties of applications using SGX enclaves. Moat models a havocing adversary who is able to modify and observe all non-enclave memory, similar to our active attacker. The key theorem verified in their work also shows that well-typed programs guarantee confidentiality of enclave code and data, which is analogous to our Theorem 1. Moat’s enclave model is specific to the Intel SGX and the x86 instruction set model, while we use Gollamudi and Chong’s [6] more abstract model. The key difference between our work and Moat is that we extend our confidentiality guarantees to an enclave-agnostic language through a verified translation. This means that programmers need not be aware of the instruction-level details of enclaves to write secure code.

Ferraiuolo et al. [5] use SecVerilog [15], a hardware description language which verifies security properties of hardware, to verify the implementation of a processor with TrustZone-like [1] enclave support. This work is complementary to ours: we assume that memory enclaves function according to our specification.

6.2 Verification of information flow properties

There is a large body of work on language-based approaches to information-flow control [11]. More recent work has used machine-checked proofs both to verify information flow properties of existing systems and to create new systems with verified information-flow guarantees. Hedin and Sabelfield [7] prove information-flow security for a core subset of JavaScript and formalize part of their proof in Coq [8]. Schwarz et al. [12] present a framework for automatically verifying non-inference at the instruction level. Swamy et al. [14] propose a new, security-typed language, FINE, and use Z3 [3] to prove the soundness of the type system.

Like Moat [13], an important contribution of our work is that we extend verified information-flow guarantees to a setting where infrastructure such as the operating system or virtual machine monitor is untrusted.

7. Future Work

Future work includes computational type-checkers for both IMPE and IMS, and a proof that these computational implementations adhere to our specification and therefore ensure the security properties proven. Future work can also extend our security proof to the full IMPE model rather than just our SIMPE calculus. Combining a proof of IMPE type-soundness with the proof for IMS to IMPE translation will lead to a proof that a well-typed IMS program, translated to a well-typed IMPE program, is secure.

Far-future work includes verifying a security-aware compiler pass for CompCert [9] that automatically inserts enclaves with a given security policy. This will require a significantly more complex model than our model of simple calculi such as IMS and IMPE. Combining this with verification of enclave hardware such as done by Ferraiuolo et al. [5] would provide end-to-end security guarantees from a high-level language to the hardware implementation.

8. Acknowledgements

We would like to acknowledge Eddie Kohler for his constant support and guidance throughout the development of this project.
References


A. Additional Figures

Cstep2.call.div:
   exp.step2 md (e,r,m)
       (VPair (Vlambda (left) (right)) (Vlambda (left) (right))) ->
   com_step
       (left, project r left, project m left)
       (right, project r right, project m right)
   com_step md
       (merge(left, right), merge(left, right))

Figure 10: The while-div call-step judgment.

Inductive exp : Type :=
   | Enat : nat -> exp
   | Evar : var -> exp
   | Ebinop : exp -> exp -> (nat -> nat) -> exp
   | Elabel : loc -> exp
   | Ederef : exp -> exp
   | Eisunset : condition -> exp
   | Elambda : seq -> exp
   with com : Type :=
       | Cskip : com
       | Cassign : var -> exp -> com
       | Cdeclassify : var -> exp -> com
       | Cupdate : exp -> exp -> com
       | Coutput : exp -> sec_level -> com
       | Ccall : exp -> com
       | Cset : condition -> com
       | Cif : exp -> seq -> seq -> com
       | Cwhile : exp -> seq -> com
   with seq : Type :=
       | Cseq : list com -> seq.

Figure 11: The IMPS grammar.

Inductive process.kill:
   set enclave -> set enclave -> list E.com ->
   list (set enclave) -> Prop :=
   | PKnil : forall K2, process.kill K2 [] [] [K2]
   | PKcons : forall K2 K3 kcoms Ksout, process.kill (K :: K2) K3 kcoms
       ((K :: K2) :: Ksout) ->
       process.kill K2 (K :: K3) (E.Ckill K :: kcoms)
       (K2 :: (K :: K2) :: Ksout).

Figure 12: The process.kill inductive type.

Inductive process_seq (ecoms : list E.com)
   (md : E.mode) (mds : list E.mode)
   (K1s K2s K3s : list (set enclave))
   (ecoms' : list E.com) (Ksout : list (set enclave)) :=
   | PS00 : Forall (fun msi => msi = md) mds ->
       process_seq ecoms md mds K1s K2s K3s ecoms' Ksout
   | PS01 : md = None ->
       mds = None :: mds' ->
       ecoms = c :: ecoms'' ->
       K1s = K1 :: K1' :: Ks1' ->
       K2s = K2 :: Ks2' ->
       K3s = K3 :: Ks3' ->
       process_kill K2 K3 kcoms pk.Ks ->
       process_seq ecoms'' md mds' (K1' :: Ks1') Ks2' Ks3'
       ecoms''(K1' :: Ks1') Ks2' Ks3'
       ecoms' = c :: kcoms ++ ecoms'' ->
       Ksout = K1 :: pk.Ks ++ Ksout' ->
       process_seq ecoms md mds K1s K2s K3s ecoms' Ksout
   | PS02 : md = None ->
       mds = mds1 ++ mds2 ->
       Forall (fun msi => msi = Encl j) mds1 ->
       hd mds2 /= Encl j ->
       ecoms = ecoms1 ++ ecoms2 ->
       length ecoms1 = length mds1 ->
       K1s = K1 :: K1s' ++ K1' :: Ks'1'' ->
       K2s = K2s' ++ K2 :: Ks'2'' ->
       K3s = K3s' ++ K3 :: K3s'' ->
       length (K1 :: K1s') = length mds1 ->
       length (K2s' ++ K2) = length mds1 ->
       length (K3s' ++ K3) = length mds1 ->
       c = E.Encl j (E.Cseq ecoms1) ->
       process.kill K2 K3 kcoms pk.Ks ->
       process_seq ecoms2 md mds2 (K1' :: Ks1'') Ks2' Ks3' ecoms''(K1' :: Ks1') Ks2' Ks3'
       ecoms' = c :: kcoms ++ ecoms'' ->
       Ksout = K1 :: pk.Ks ++ Ksout' ->
       process_seq ecoms md mds K1s K2s K3s ecoms' Ksout.

Figure 13: The process_seq inductive type.