Better Trust Zone: Verifying Security of Enclave-Aware Calculi

Aaron Bembenek 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]

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);

[1] 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 Gollamundi 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* $\mathscr{L} = \{L, H, \top\}$. 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 learn.

A *security policy* is either a *security level policy* ℓ where $\ell \in \mathscr{L}$ or an *erasure policy* $\ell_1 \nearrow^{cnd} \ell_2$ where $\ell_1, \ell_2 \in \mathscr{L}$. 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 *cnd* is met. For example, the erasure policy $H \nearrow^{done} \top$ means that information is privileged until condition *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 `exp` and `com` for IMPE expressions and commands.

```
Inductive mode : Type :=
| None : mode
| Encl : enclave → mode.

Inductive exp : Type :=
| Enat : nat → exp
| Evar : var → exp
| Ebinop : exp → exp → (nat → nat → nat) → exp
| Eloc : location → exp
| Ederef : exp → exp
| Eisunset : condition → exp
| Elambda : mode → com → 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
| Cenclave : enclave → com → com
| Ckill : enclave → com
| Cseq : list com → com
| Cif : exp → com → com → com
| Cwhile : exp → com → com.

Inductive val : Type :=
| Vlambda : mode → com → val
| Vnat : nat → val
| Vloc : location → val.

Definition mem : location → val.
Definition register : variable → val.
```

Figure 1: IMPE *syntax.*

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-

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` $\rightarrow$ `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 `loc` which is in an enclave (that is, δ`(loc) = Encl i` for some `i`).

```
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
```

Figure 2: IMPE *semantic judgments.*

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 `loc` must have the type Σ(`loc`). Neither Γ nor Σ can ever map to a type with security level $\top$. 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 `com_wt` takes as an argument a set of conditions that are unset when the command `com` is executed. This is necessary to determine the current security level of each location, since a location may have a erasure security policy.

```
Inductive impe_type := Tnat | Tlambda | Tloc
Inductive exp_wt : var_context → exp →
                   impe_type → Prop
Inductive com_wt : mode → policy → var_context →
                   set condition → com →
                   var_context → Prop
```

Figure 3: IMPE *expression and command typing judgments*

$$\begin{array}{l}\texttt{exp\_wt}\ \Gamma\ \texttt{e}\ \sigma_p \rightarrow \\ \texttt{com\_wt md pc}_b\ \Gamma\ \texttt{unsetCnds c}\ \Gamma' \rightarrow \\ \texttt{pc}_b \geq \texttt{p} \wedge \texttt{pc}_b \geq \texttt{pc} \rightarrow \\ \texttt{com\_wt md pc}\ \Gamma\ \texttt{unsetCnds (Cwhile e c)}\ \Gamma'.\end{array}$$

Figure 4: The typing judgment for a `Cwhile e c` *command that prevents leaks through implicit flows.*

```
if (*password == *guess) output 1 to Low
else output 0 to Low
```

Figure 5: A program with implicit flows that is not well-typed when $\Sigma(password) = Tnat_H$ *and* $\Sigma(guess) = Tnat_H$.

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 (`Cwhile` or `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 `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 $\Sigma(\texttt{password}) = Tnat_H$ and $\Sigma(guess) = Tnat_H$. The expression `*password == *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 `kill` command for killing an enclave during execution.
3. SIMPE adds the restriction that the initial memory m_{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.

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_{left}, is the initial memory upon which the attacker observes the SIMPE program execution. The second memory m_{right} is any memory that must be in the attacker's entropy set in order for the program to be secure.

```
Inductive val2 : Type :=
| VSingle : val → val2
| VPair : val → val → val2
Definition mem2 := location → val2
Definition reg2 := variable → val2
Definition trace2 (t1 t2: trace) := merge(t1, t2)

Definition exp_config2 := exp * reg2 * mem2
Definition com_config2 := com * reg2 * mem2
Definition final_config2 := reg2 * mem2

Inductive exp_step2 : mode → exp_config2 →
                      value → Prop
Inductive com_step2 : mode → com_config2 →
                      final_config2 → trace2 →
                      Prop
```

Figure 6: SIMPE^2 *syntax definitions and large-step judgment.*

By tracking certain invariants of the SIMPE^2 execution, we prove that the execution of any well-typed SIMPE program on m_{left} produces the same observable output as the program's execution on m_{right}. This ensures that the attacker must believe memory m_{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 memories m_{left} and m_{right} simultaneously, where m_{left} and m_{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_{left}, m_{left}, and t_{left}, and the register and memory of the second execution as r_{right}, m_{right}, and t_{right}. We *merge* r_{left} and r_{right} to get a `reg2`, and can *project* a `reg2` value to get back the values of r_{left} and r_{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, Σ(loc) = σ_L → 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 SIMPE^2.

```
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) = σ_sl
          → sl = H ∧ md ≠ None)
    ∧ (∀ loc, m(loc) = VPair v1 v2 ∧ Σ(loc) = σ_sl
          → sl = H ∧ δ(loc) ≠ None).

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

Figure 8: Typing judgments for SIMPE^2 *configurations and final configurations.*

dividing executions. A dividing execution is when the two SIMPE executions modeled by SIMPE^2 execute different branches of a conditional, or different function calls. This occurs when the expression `e` in a `(while e do c)`, `(if e then c1 else c2)`, or `call(e)` command evaluates to a `VPair`. Figure 10 in the Appendix shows an example of our `call-div` large-step judgment.

The SIMPE^2 typing judgments are the same as SIMPE typing judgments.

With SIMPE^2, we can state the definition of a secure program (Figure 7), where m_{left} and m_{right} are the memories described in Section 3.1.

3.3 SIMPE^2 Adequacy

We first prove the adequacy of SIMPE^2: SIMPE^2 is both sound and complete. This allows us to extend our reasoning about the pairs of execution traces and values produced by a SIMPE^2 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 SIMPE^2 large-step execution corresponds to two individual SIMPE large-step executions. The proof of soundness follows by straightforward induction on the SIMPE^2 large-step derivation.

LEMMA 2. *(Completeness)*

```
com_step md (c, rleft, mleft)
   (rleft′, mleft′) tleft →
com_step md (c, rright, mright)
   (rright′, mright') tright →
com_step2 md
   (c, merge(rleft, rright), merge(mleft, mright))
   (merge(rleft′, rright′), merge(mleft′, mright′))
    merge(tleft′, tright′).
```

Completeness ensures that any two SIMPE large-step executions can be represented as a single SIMPE^2 large-step execution by merging configurations together. The proof of completeness follows by induction on one of the SIMPE large-step derivations, and inversion on the other large-step derivation.

3.4 SIMPE^2 Well-Typed Configuration Invariants

We need to show that SIMPE^2 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 SIMPE^2 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:

(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.

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)*

```
config2_wt md pc Γ cconfig2 Γ' →
com_step2 md cconfig2 finalcconfig2 t2 →
finalconfig2_wt md pc Γ finalcconfig2.
```

LEMMA 4. *(Intermediate Configuration Invariant Preservation)*

```
config2_wt md pc Γ cconfig2 Γ' →
com_step2 md cconfig2 finalcconfig2 t2 →
intermediate_config (cconfig2_im) (cconfig2) →
com_step2 md (cconfig2_im) (finalcconfig2_im) (t2_im) →
∃ pc_im, Γ_im, Γ'_im,
    config2_wt md pc_im Γ_im cconfig2_im Γ'_im ∧ pc_im ≥ pc.
```

The proof of both Lemma 3 and Lemma 4 proceeds by induction on the large-step semantics (`com_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 `VPair`, `e` is typed with high security and executes in an enclave.

LEMMA 5. *(Expression Pair Invariants)*

```
exp_step2 md (e, r, m) (VPair v1 v2) →
val_wt Γ e σ_sl →
sl = H ∧ md ≠ 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 `val_wt` is defined similarly to `exp_wt`.

LEMMA 6. *(Value Type Preservation)*

```
exp_step2 md (e, r, m) v ∧ exp_wt Γ e σ_sl →
val_wt Γ v σ_sl.
```

Lemma 5 allows us to verify that an assignment of a `VPair` value to a register, or an update of a memory location with a `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 `update(loc,e)` or in `assign(x,e)` evaluates to a `VPair`. Lemma 5 ensures that the value put at location `loc` or assigned to `x` is high-security, and the update/assignment occurs in an enclave.
- `e` in `(while e do c)`, `(if e then c1 else c2)`, or `call(e)` command evaluates to a `VPair`. Lemma 5 ensures that the type of `e` is high security. As described in Section 2.3, the type system ensures that the subcommands must all be executed with security at least as high as that of `e` and in the same mode as `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 `VPair` values into registers or memory. The only other way that a register or memory location can be modified is via the `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 `VPair`, and therefore a `declassify` command cannot introduce a `VPair` into a register.

Because registers and memories gain `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_{right} is any memory that must be in the attacker's entropy set.

LEMMA 7. *(Observational Equivalence)*

```
config2_wt md pc Γ cconfig2 Γ' →
com_step2 md cconfig2 final_config2 t2 →
observations_low_channel (project_trace t2 left) =
observations_low_channel (project_trace t right).
```

The proof of observational equivalence proceeds by induction on the large-step semantics (`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_{left} are equivalent to the observable effects of the program executing on m_{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 $sl \neq L$. Otherwise, the value output to sl is not a `VPair`, and the attacker observes either no value if $sl = H$ 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 SIMPE[2] diverges on m_{left} compared to m_{right}, the execution produces no observable differences.

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

```
com_wt md pc Γ c Γ' ∧ pc = H →
com_step2 md (c, r, m) (r', m') t2 →
observations_low_channel t2 = [].
```

3.7 Noninterference Security

From observational equivalence, our final security theorem follows: a well-typed SIMPE 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)*

```
com_wt None L Γ c Γ' → command_secure c.
```

If `c` is well-typed, then the configuration `(c, reg_init2, mem_init2)`, where `reg_init2` is a pair of registers initialized to all 0 values, and `mem_init2 = merge(mleft, mright)` is well-typed:

```
config2_wt None L Γ (c, reg_init2, mem_init2) Γ'
```

Lemma 7 asserts that the execution on m_{left} produces the same observable output as the execution on m_{right}. We therefore conclude that m_{right} is in the attacker's entropy set. Thus, a well-typed 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_{left} than when it executes on m_{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` preservation ensures if execution on m_{left} ever contains a value in a register or in memory that differs from that of the execution on m_{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_{left} from an execution on m_{right}, because an active attacker cannot access memory in an enclave, nor modify enclave code.

4. Security Verification Experience

When specifying the IMPE and SIMPE calculi, we make slight modifications to Gollamudi and Chong's original IMPE model to aid verification. These modifications follow a common theme of making implicit assumptions explicit.

We make explicit that the location context Σ is constant through execution. We introduce Σ as a global parameter to the model, rather than threading the context through the program's typing judgment.[2] By explicitly specifying that Σ does not change, we are able to eliminate concepts from the original model, such as the secu-

[2] Gollamudi and Chong have a single typing context for both variables and locations. We separate this context into two contexts, Γ and Σ, 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 $\top$) also allows us to ignore killed enclaves: killed enclaves allow us to have values typed with security $\top$, 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 `com_wt` produces only one possible constructor.

AXIOM 1. *(Subsumption Typing Rule)*

$$\texttt{com\_wt md pc1 } \Gamma_1 \texttt{ c } \Gamma_1' \rightarrow \texttt{pc2} \leq \texttt{pc1} \wedge \Gamma_2 \leq \Gamma_1 \wedge \Gamma_1' \leq \Gamma_2' \rightarrow \texttt{com\_wt md pc2 } \Gamma_2 \texttt{ c } \Gamma_2'.$$

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 `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)*

$$\forall \texttt{ loc, m(loc)} \neq \texttt{Vloc loc}'$$

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)*

$$\begin{aligned}&\forall \texttt{ loc, match minit l with}\\ &\quad\texttt{| Vlambda c} \Rightarrow \Sigma(\texttt{l}) = \texttt{Tlambda}_{sl} \wedge \\ &\qquad\qquad\texttt{com\_wt md pc } \Gamma_m \texttt{ c } \Gamma_p\\ &\quad\texttt{| Vloc l} \Rightarrow \texttt{False}\\ &\quad\texttt{| Vnat n} \Rightarrow \Sigma(\texttt{l}) = \texttt{Tnat}_{sl}\\ &\texttt{end.}\end{aligned}$$

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)*

$\forall$ reg, $\forall$ mem,
com_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 Elambda or Eloc 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)*

$$\texttt{reg(x)} = \texttt{Vlambda c} \wedge \Gamma(\texttt{x}) = \texttt{Tlambda}_{sl} \leftrightarrow \texttt{exp\_type md}' \ \Gamma \ (\texttt{Elambda c}) \ \texttt{Tlambda}_{sl'}$$

$$\texttt{reg(x)} = \texttt{Vloc loc} \wedge \Gamma(\texttt{x}) = \texttt{Tloc}_{sl} \leftrightarrow \texttt{exp\_type md}' \ \Gamma \ (\texttt{Eloc loc}) \ \texttt{Tloc}_{sl'} \wedge \delta(\texttt{l}) = \texttt{md'}$$

$$\texttt{mem(loc)} = \texttt{Vlambda c} \wedge \Gamma(\texttt{x}) = \texttt{Tlambda}_{sl} \leftrightarrow (\texttt{exp\_type md}' \ \Gamma \ (\texttt{Elambda c}) \ \texttt{Tlambda}_{sl'} \vee (\texttt{mem\_init(loc)} = \texttt{Vlambda c}))$$

Axiom 5 follows from Axiom 4. We know that any function or location value comes from one of three sources: (1) an expression Elambda or Eloc, (2) another register, or (3) a memory location. We know that the initial registers start out with value 0, and the initial memory is 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 location must have come from one of the same 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 Elambda or Eloc.

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 com_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 reg_init and 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)*

$$\texttt{v} = \texttt{VPair v1 v2} \leftrightarrow \texttt{v1} \neq \texttt{v2} \wedge ((\texttt{v1} = \texttt{Vnat n1} \wedge \texttt{v2} = \texttt{Vnat n2}) \vee (\texttt{v1} = \texttt{Vlambda c1} \wedge \texttt{v2} = \texttt{Vlambda c2})).$$

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., when $\Gamma_1 \leq \Gamma_2$). We are able to prove Lemma 3 using this definition of the subtyping relation:

$$\forall \texttt{x}, \Gamma_1(\texttt{x}) = \sigma_{sl} \wedge \Gamma_2(\texttt{x}) = \sigma_{sl'} \rightarrow sl \leq sl'$$

Gollamudi and Chong require that the domains of Γ_1 and Γ_2 be equivalent, whereas we require only that the domain of Γ_2 be *at least as large* as that of Γ_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 while-div rule for SIMPE^2. Gollamudi and Chong state that a (while e do c) command is equivalent to executing either a skip command or the command c. This does not allow a 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 `com_step2` instead of performing dual induction on two `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 `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 `seq`, instead of an arbitrary command, and a top-level IMPS program is also a `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:

```
exp_wt : exp → type → deriv → 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 `S` and the latter from module `E` (e.g., `S.exp_wt` versus `E.exp_wt`).

```
com_wt : com → deriv → Prop
```

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:

```
exp_wt e t drv →
com_wt (S.Coutput e chan) (Deriv t drv)
```

The `Deriv t drv` term represents that during the derivation of the typing judgment, the subexpression `e` had type `t` and derivation `drv`, which in turn constrains the hypotheses we get by inverting on the typing judgment. Although this was not part of the model of Gollamudi and Chong, we found that it was essential to explicitly pass derivation information between typing judgments and translation judgments to prove that the translation scheme preserves well-typedness.

5.2 Modeling the translation scheme

The goal of the translation scheme is to ensure that any translation that meets its constraints produces well-typed (and therefore secure) IMPE code. Intuitively, this means guaranteeing that if a translation satisfies the constraints of the scheme, then it correctly places sensitive code and memory in enclaves. The constraints for the translation scheme are modeled via three mutually inductive types: `exp_trans`, `com_trans`, and `seq_trans`. An `exp_trans` judgment has the form

```
exp_trans e t drv md δ e′ t′
```

and indicates that the IMPS expression `e` with type `t` and derivation `drv` translates to the IMPE expression `e′` with type `t′`. Additionally, the translation constrains the mode (`md`) for the expression (which determines whether the expression needs to run in a particular enclave) and the distribution of memory locations between normal memory and enclaves (δ).

A command translation judgment in the form

```
com_trans c drv md δ K1 c′ K2
```

represents that IMPS command `c` with derivation `drv` translates to IMPE command `c′`, where `md` and δ are defined as in the translation judgment for expressions. `K1` is the set of killed enclaves before `c′` and `K2` is the set of killed enclaves after `c′`.

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

```
exp_trans e t drv md δ e′ t′ ∧ ∼In md K →
com_trans (S.Coutput e chan) (Deriv t drv) md δ
    K (E.Coutput e′ chan) 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 (`scoms`), a list of IMPE commands (`ecoms`), 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 $\texttt{mds}_i$ with initial kill set $\texttt{K1}_i$ and finishes with kill set $\texttt{K2}_i$.
2. $\texttt{K1}_{i+1}$ is the union of $\texttt{K2}_i$ and $\texttt{K3}_i$. Since $\texttt{K2}_i$ is the kill set immediately after $\texttt{ecoms}_i$ finishes and $\texttt{K1}_{i+1}$ is the kill set immediately before $\texttt{ecoms}_{i+1}$, this premise allows the translation to insert `kill` commands between $\texttt{ecoms}_i$ and $\texttt{ecoms}_{i+1}$ that destroy the enclaves in $\texttt{K3}_i$, a set which is underspecified (although subject to restrictions given in the following premises).
3. $\texttt{K2}_i$ and $\texttt{K3}_i$ 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`, and 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

```
Inductive seq_trans := TRseq :
  (∀ 0 ≤ i < length scoms,
    com_trans scoms[i] drvs[i] mds[i] δ K1s[i]
      ecoms[i] K2s[i] ∧
    K1s[i + 1] = K2s[i] ∪ K3s[i] ∧
    K2s[i] ∩ K3s[i] = ∅ ∧
    (md ≠ None → mds[i] = md) ∧
    (mds[i] ≠ None ∧ mds[i] = mds[i+1] →
      K3s[i] = ∅) ) →
  process_seq ecoms md mds K1s K2s K3s ecoms' Ksout →
  seq_trans (S.Cseq scoms) (Derivseq drvs) md δ
      (hd Ksout) (E.Cseq ecoms') (last Ksout)
```

Figure 9: The unique constructor for the `seq_trans` *type.*

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

```
process_seq ecoms md mds K1s K2s K3s ecoms' Ksout
```

is constructed in one of three ways (see Figure 13 in the Appendix for the pseudocode):

1. The first constructor applies when every mode in `mds` is the same as `md` and the kill sets in `K3s` are all empty. This just "returns" the original sequence of commands `ecoms`. This is the only constructor that applies when `md` is not normal.
2. The second constructor applies when `md` is `None` and the mode of the first command `c` of `ecoms` is also `None`. It constructs the output sequence of commands `ecoms'` by sticking together `c`, `kill` commands that kill the enclaves specified by the first member of `Ks3`, and the result of recursing on the remaining commands.
3. The third constructor applies when `md` is `None` and the first *m* commands in `ecoms` need to run in some particular enclave. It places these commands in the appropriate enclave, performs the kills specified by the *m*th member of `Ks3`, and then recurses on the remaining commands.

The constructors also refer to a helper `process_kill` (Figure 12 in the Appendix). If the judgment

```
process_kill K2 K3 kcoms Ksout
```

holds, then `kcoms` is a sequence of `kill` commands destroying the enclaves in the kill set `K3`, and `K2` is the set of enclaves that are already killed. Once again, the `Ksout` term is included to track the kill sets before and after each `kill` command in sequence `kcoms`.

5.3 Translation Verification

The ultimate goal is to prove that every translation that meets the constraints of the translation scheme produces a well-typed ImpE program when given a well-typed ImpS program:

Theorem 2. *(Translation Soundness)*

```
S.seq_wt c drv →
seq_trans c drv md δ K1 c' K2 →
E.com_wt md δ K1 c' K2.
```

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)*

```
(∀ 0 ≤ i < length ecoms,
    E.com_wt md δ K1s[i] ecoms[i] K2s[i]) →
... (* premises from seq_trans *) ...
process_seq ecoms md mds K1s K2s K3s ecoms' Ksout →
E.com_wt md δ (hd Ksout) (E.Cseq ecoms') (last 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 ∩ K3 = ∅ ∧ process_kill K2 K3 kcoms Ksout →
∀ 0 ≤ i < length kcoms,
    E.com_wt md δ Ksout[i] kcoms[i] Ksout[i+1].
```

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

```
(∀ 0 ≤ i < length ecoms,
    E.com_wt md δ K1s[i] ecoms[i] K2s[i]) →
... (* premises from seq_trans *) ...
process_seq ecoms md mds K1s K2s K3s ecoms' Ksout →
∀ 0 ≤ i < length ecoms',
    E.com_wt md δ Ksout[i] ecoms'[i] Ksout[i+1].
```

The `Ksout` term is key to both these lemma statements. The claim we are making is that, in both the case of `process_kill` and `process_seq`, the ith command in the output sequence is well-typed when preceded by the ith kill set of `Ksout` and followed by the $(i+1)$th kill set of `Ksout`. While the definitions of `process_seq` and `process_kill` do not require such a `Ksout` 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 `K2` and `K3` are disjoint and the definition of `process_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 `process_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 `process_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 IMPS, 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 IMPS to IMPE translation will lead to a proof that a well-typed IMPS 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 IMPS 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

[1] ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[2] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] R. DeLine and K. R. M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. 2005.

[5] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a practical hardware security architecture. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[6] A. Gollamudi and S. Chong. Automatic enforcement of expressive security policies using enclaves. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2016. ACM Press.

[7] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 3–18. IEEE, 2012.

[8] INRIA. Coq. `https://coq.infria.fr`. Accessed: 2017-05-04.

[9] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[10] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, 2003.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[12] O. Schwarz and M. Dam. Automatic derivation of platform noninterference properties. In *International Conference on Software Engineering and Formal Methods*, pages 27–44. Springer, 2016.

[13] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184. ACM, 2015.

[14] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *European Symposium on Programming*, pages 529–549. Springer, 2010.

[15] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 503–516. ACM, 2015.

A. Additional Figures

```
Cstep2_call_div:
  exp_step2 md (e, r, m)
      (VPair (Vlambda cleft) (Vlambda cright)) →
  com_step md
      (cleft, project r left, project m left)
       (rleft, mleft) tleft →
  com_step md
       (cright, project r right, project m right)
       (rright, mright) tright →
  com_step2 md (c, r, m)
       (merge(rleft, rright), merge(mleft, mright))
       (merge(tleft, tright))
```

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

```
Inductive exp : Type :=
| Enat : nat → exp
| Evar : var → exp
| Ebinop : exp → exp → (nat → nat → nat) → exp
| Eloc : location → 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 K 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)) :=
| PSO0 :
    Forall (fun mdi ⇒ mdi= md) mds →
    Forall (fun K3 ⇒ K3 = []) K3s →
    ecoms = ecoms′ →
    Ksout = (hd Ks) :: K2s →
    process_seq ecoms md mds K1s K2s K3s
        ecoms′ Ksout
| PSO1:
    md = None →
    mds = None :: mds′ →
    ecoms = c :: ecoms″ →
    K1s = K1 :: K1′ :: K1s′ →
    K2s = K2 :: Ks2′ →
    K3s = K3 :: Ks3′ →
    process_kill K2 K3 kcoms pk_Ks →
    process_seq ecoms″ md mds′
        (K1′ :: K1s′) Ks2′ Ks3′
        ecoms′ (K1′ :: Ksout′) →
    ecoms′ = c :: kcoms ++ ecoms″ →
    Ksout = K1 :: pk_Ks ++ Ksout′ →
    process_seq ecoms md mds K1s K2s K3s
        ecoms′ Ksout
| PSO2:
    md = None →
    mds = mds1 ++ mds2 →
    Forall (fun mdi ⇒ mdi = Encl j) mds1 →
    hd mds2 ≠ Encl j →
    coms = coms1 ++ coms2 →
    length coms1 = length mds1 →
    K1s = K1 :: K1s′ ++ K1′ :: K1s″ →
    K2s = K2s′ ++ K2 :: K2s″ →
    K3s = K3s′ ++ K3 :: K3s″ →
    length (K1 :: K1s′) = length mds1 →
    length (K2s′ ++ [K2]) = length mds1 →
    length (K3s′ ++ [K3]) = length mds1 →
    c = E.Cenclave j (E.Cseq coms1) →
    process_kill K2 K3 kcoms pk_Ks →
    process_seq coms2 md mds2 (K1′ :: K1s″)
        K2s′ K3s′ ecoms″ (K1′ :: Ksout′) →
    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.