Automatic Enforcement of Expressive Security Policies using Enclaves

Anitha Gollamudi Stephen Chong
Harvard University, Cambridge, MA, USA
agollamudi@g.harvard.edu chong@seas.harvard.edu

Abstract

Hardware-based enclave protection mechanisms, such as Intel's SGX, ARM's TrustZone, and Apple's Secure Enclave, can protect code and data from powerful low-level attackers. In this work, we use enclaves to enforce strong application-specific information security policies.

We present IMPE, a novel calculus that captures the essence of SGX-like enclave mechanisms, and show that a security-type system for IMPE can enforce expressive confidentiality policies (including erasure policies and delimited release policies) against powerful low-level attackers, including attackers that can arbitrarily corrupt non-enclave code, and, under some circumstances, corrupt enclave code.

We present a translation from an expressive security-typed calculus (that is not aware of enclaves) to IMPE. The translation automatically places code and data into enclaves to enforce the security policies of the source program.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

Keywords Enclave programs, information erasure, declassification, security-type system, information-flow control, language-based security.

1. Introduction

Language-based techniques for security can enforce expressive information security policies for applications. Enforceable policies include ensuring that application-level adversaries learn nothing about confidential information [29, 38], that some clearly specified confidential information may be released under controlled circumstances [31], and that sensitive information is correctly removed from the system at appropriate times [7, 10]. However, these language-based guarantees may fail to hold in the presence of low-level attackers, such as attackers that observe execution at the level of operating-system or hardware abstractions, or attackers that can inject arbitrary code into a process.

Recent hardware-based *enclave protection mechanisms* (including Intel's SGX [25], ARM's TrustZone [4], and Apple's Secure Enclave [2]) can protect code and data from low-level attacks, including compromised kernels. These new mechanisms present an opportunity to extend strong application-specific information security guarantees to hold against low-level attackers.

We take advantage of this opportunity: we present a language model that captures the essence of enclave protection mechanisms, and give a security-type system for this language that enforces strong non-interference-based information security guarantees [12, 18], including delimited release [30] and information erasure [9]. Moreover, we provide a translation from a non-enclave source language that automatically infers which code and data to place in enclaves in order to enforce expressive security policies.

As an example of application-specific information security requirements, consider code that authenticates a user. The user provides a guess that is checked against the actual password. If the guess matches the password, the user is authenticated and the computation continues. After authentication, the guess is no longer needed, and the subsequent computation should in no way depend on the guess. This information security requirement can be expressed as an *erasure policy* [9] that requires restrictions on the use of sensitive information (i.e., the user's guess) after certain conditions are satisfied (i.e., the user is successfully authenticated). Language-based techniques can ensure that these restrictions are respected by the subsequent computation (e.g., [7, 24]).

However, these techniques typically enforce security against a language-level attacker that passively observes the program's output or perhaps provides code that is subject to similar enforcement mechanisms as the program itself (e.g., [3, 26]). The desired security guarantees may fail to hold in settings where an attacker has privileged access to

OOPSLA'16 October 25–30, 2016, The Netherlands

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00

a machine (such as in cloud services or on mobile devices) or an attacker is able to exploit vulnerabilities to observe more data than anticipated (such as in the Heartbleed attack) or inject arbitrary code into the program's process (such as in buffer overrun attacks). In these cases, an attacker that compromises the system sometime after the user has authenticated may be able to learn the user's password. For example, even though the program may not in any way use the user's guess in the subsequent computation, the bits representing the guess may still be present in physical or virtual memory, and accessible to a low-level attacker [21, 22].

Enclave protection mechanisms can secure code and data against powerful attackers, including malicious code within the same process or a malicious operating system. Intel's SGX extends the x86 instruction set with additional instructions that allow a contiguous region of memory within a process's address space to be established as an enclave, and subsequently uses hardware-enforced access control to ensure that code outside of an enclave is unable to access data within an enclave. Moreover, execution may enter an enclave only via specified entry points. Memory within an enclave is encrypted before being paged out.

But leveraging enclaves to enforce application-specific information security guarantees is hard. Enclave mechanisms place the onus on the programmer to secure an application by effectively decoupling the security-critical parts of the application from the non-critical and/or untrusted parts of the application. Hardening an application to carefully isolate the dependencies requires non-trivial effort [32, 35].

In this paper we consider the automatic enforcement of application-specific information security policies using enclaves. We make several contributions.

1. We present IMPE, a novel calculus that captures the essence of SGX-like enclave mechanisms (Section 2).
2. We show that a security-type system for IMPE can enforce expressive confidentiality policies (including erasure policies and delimited release policies) against several attackers, including attackers that can arbitrarily corrupt non-enclave code, and, under certain circumstances, corrupt enclave code (Sections 3 and 4).
3. We present a translation from a non-enclave source language to IMPE (Sections 5 and 6). The programmer can focus on the correct handling of information in the source language, and the translation will automatically infer appropriate placement of code and data into enclaves to ensure security guarantees against powerful low-level attackers. The translation can be configured to optimize various criteria, including reducing the size of the trusted computing base, reducing the runtime performance impact of using enclave mechanisms, or removing erased data as soon as possible.

In addition, we validate the translation and the expressiveness of IMPE by implementing several simple models of applications with application-specific security guarantees (Section 8).

2. IMPE: a Calculus for Enclaves

We present IMPE, an imperative higher-order calculus that captures the key features of enclaves and moreover supports the specification of information security policies, including policies for information erasure.

2.1 Security Levels and Policies

We use a set of *security levels* $\mathcal{L} = \{L, H, \top\}$ to express confidentiality restrictions on information. Security level L ("low security") is for public information that anyone, including an attacker, is permitted to learn. Security level H ("high security") is for confidential information that only trusted entities are permitted to learn. Security level $\top$ is for information so confidential that no-one is permitted to learn it. Ideally, the system never contains information with security level $\top$.

Let partial order $\sqsubseteq$ be the smallest reflexive and transitive relation such that $L \sqsubseteq H$ and $H \sqsubseteq \top$. Intuitively, if $\ell_1 \sqsubseteq \ell_2$, then information with security level ℓ_2 is at least as confidential as information with security level ℓ_1. Security levels ordered by $\sqsubseteq$ form a lattice.

The security level to enforce on information may change over time. In this paper, we focus on *information erasure*: the requirement that when a specific condition is met, information needs to become more confidential.

Security policies describe how the security level of information must change over time. A *security level policy* ℓ simply means that information must be handled with security level ℓ at all times. An *erasure policy* $\ell_1 \overset{cnd}{\nearrow} \ell_2$ means that initially information can be handled according to security level ℓ_1. However, when condition cnd is met, the information must be handled according to security level ℓ_2, where $\ell_1 \sqsubseteq \ell_2$. *Conditions* are used to express when information must be "erased" or made more restrictive. In general, conditions for erasure can be arbitrary state predicates [9]. However, we encode conditions using mutable memory locations: a condition cnd is represented by a single memory location, and the condition is regarded as satisfied exactly when the location contains a non-zero value. The program is responsible for setting the condition location to a non-zero value to correctly reflect the intended meaning of the condition. Once set (i.e., assigned a non-zero value), we do not allow a condition to be unset. This approach is sufficiently expressive and simplifies specification and reasoning about erasure policies [7].

We write P to denote the set of policies, and use metavariables p, q to range over policies. We refer to any information labeled with a policy more restrictive than L as confidential information.

Consider a program that authenticates a password. Let password be a memory location that stores password input

$$
\begin{aligned}
e &::= n \mid x \mid e_1 \oplus e_2 \mid l \mid *e \mid \mathsf{isunset}(cnd) \mid \lambda^{\mu}.c \\
v &::= \lambda^{\mu}.c \mid n \mid l \\
c &::= \mathsf{skip} \mid x := e \mid x := \mathsf{declassify}(e) \mid e_1 \leftarrow e_2 \\
&\quad \mid \mathsf{output}\ e\ \mathsf{to}\ \ell \mid \mathsf{call}(e) \mid \mathsf{set}(cnd) \mid \mathsf{enclave}(i, c) \\
&\quad \mid \mathsf{kill}(i) \mid c_1; \ldots; c_n \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
l &\in Loc \qquad cnd \in Cond \qquad Cond \subset Loc \\
x &\in \mathit{Vars} \qquad i, n \in \mathbb{N} \\
\mu &\in Mode = \{N, E_1, E_2, \ldots\}
\end{aligned}
$$

Figure 1. IMPE Syntax

from a user. Once authentication succeeds, it is desirable to erase password entirely from the memory. If end is a condition that indicates whether the authentication session has ended, a suitable policy for password can be $L \overset{\mathsf{end}}{\nearrow} \top$. The policy says that the confidentiality level of password is initially L, and once end is set, it must be $\top$.

2.2 Syntax

IMPE is a simple imperative language. However, it includes first-class locations and functions, output commands, and models enclaves. An *enclave* consists of code and memory locations. Memory locations within an enclave can be accessed only by that enclave's code. Control can be transferred to code inside an enclave only through a predefined set of entry points. Thus, data stored inside an enclave's memory locations is protected from non-enclave code (and also from code in other enclaves). In IMPE, enclaves provide a simple yet expressive model of architectural features—such as Intel's SGX [25]—that can provide strong isolation guarantees for code and data from other code within the same process or machine.

We allow an arbitrary number of enclaves, indexed with natural numbers. We use *modes* to indicate which enclave code or data exists in, or whether it is outside of any enclave. Specifically, we use metavariable μ to range over the set $Mode = \{N, E_1, E_2, \ldots\}$, where E_i indicates the ith enclave and N indicates non-enclave (or "normal") mode.

Figure 1 shows the syntax of IMPE. Expressions e include integers n, variables x, and memory locations l. All variables have global scope. Variables are analogous to registers: they are mutable locations, but are not first-class values. By contrast, memory locations are first class, and can be passed as values. Conditions $Cond$ are a subset of the memory locations and cnd ranges over conditions. We write Loc for the set of memory locations.

Operator $\oplus$ ranges over arbitrary (total) binary operations over integers. Dereference $*e$ evaluates e to a memory location and evaluates to the contents of that location.

Expression $\mathsf{isunset}(cnd)$ tests whether condition cnd has been set, and evaluates to 1 if it is not set, 0 otherwise. Although this expression is semantically equivalent to $*cnd \neq 0$, our type system gains precision through the use of $\mathsf{isunset}(cnd)$.

Expression $\lambda^{\mu}.c$ is a first-class function. It can be thought of as a code pointer to command c. Arguments to the function are given via variables and memory locations, as are any values returned by the function. Annotation μ indicates the mode in which the function is defined. It can be thought of as indicating whether the code pointer is to an enclave or non-enclave region of memory. The annotation is used to restrict how functions can be invoked, to ensure that non-enclave code cannot enter an enclave by invoking a function that resides in the enclave.

Values v in IMPE include integers, memory locations (including conditions), and first-class functions.

Commands in IMPE include standard imperative commands (skip, $x := e$, if e then c_1 else c_2, and while e do c). We assume sequences $c_1; \ldots; c_n$ are flattened (i.e., that none of $c_1; \ldots; c_n$ are sequence commands), and for convenience assume that all sub-commands are sequences (possibly of length 1). Indirect assignment $e_1 \leftarrow e_2$ evaluates e_1 to a memory location, and updates the contents of that location with the result of e_2. We further require that e_1 does not evaluate to a condition. Command $\mathsf{set}(cnd)$ updates the contents of cnd to 1. Conditions can be updated only with a set command.

Command $\mathsf{output}\ e\ \mathsf{to}\ \ell$ evaluates e to a value and outputs it to channel ℓ. Output commands model observations by trusted and untrusted entities. We restrict ℓ to be either L or H. Intuitively, an output to channel L may be observed by an untrusted entity, such as an attacker, whereas output to channel H may be observed only by trusted entities.

Command $x := \mathsf{declassify}(e)$ is semantically equivalent to assignment $x := e$, but indicates a declassification, which is relevant for both our semantic security conditions (Section 3.2) and type system (Section 4). To simplify our semantic security condition, we require that expression e does not contain any variables (although it may contain memory locations). Command $\mathsf{call}(e)$ evaluates e to a function, and invokes the function.

Command $\mathsf{enclave}(i, c)$ defines an entry point for the enclave E_i. That is, command c is code that resides inside enclave E_i, and non-enclave code is permitted to execute c. We require that c does not contain any subcommands of the form $\mathsf{enclave}(i', c')$, i.e., enclave commands cannot be nested, regardless of whether for the same enclave or a different enclave. Commands not lexically nested in an $\mathsf{enclave}(i, \ldots)$ are non-enclave code.

We allow an enclave to have multiple entry points. That is, a program may contain multiple commands of the form $\mathsf{enclave}(i, c)$ with the same enclave identifier i.

Command $\mathsf{kill}(i)$ tears down enclave E_i. Once killed, an enclave cannot be used: its memory locations can not be accessed, nor can its code be executed.

2.3 Operational Semantics

A *configuration* $\langle c, r, m, K\rangle$ describes the current state of the system. Command c is the rest of the program to execute. Register file r maps variables to values, and memory m maps locations to values. *Kill set* K is the set of enclaves that have been killed so far in the execution.

As a program executes, it performs observable actions (i.e., outputting values on channels) and non-observable security-relevant actions (such as declassifications). We refer to these actions as *events* and use metavariable α to range over events. A *trace* $t = \alpha_1 \ldots \alpha_n$ is a finite sequence of events. We write ϵ for the empty trace, $|t|$ for the length of trace t, and $t_1 \cdot t_2$ for the concatenation of traces t_1 and t_2.

We define the semantics of IMPE with a large step operational semantics. The judgment for the evaluation of commands has the following form.

$$\mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'$$

The judgment is parameterized by mode μ, which indicates whether command c is executing in normal mode ($\mu = N$) or in an enclave ($\mu = E_i$). Initially, program execution always starts in normal mode (since all enclave code is inside $\mathsf{enclave}(i, \ldots)$ commands).

The judgment is also parameterized by function $\delta : Loc \to Mode$ which indicates for each memory location which enclave, if any, it belongs to. If $\delta(l) = E_i$ then location l is in enclave E_i, and if $\delta(l) = N$ then l is not in an enclave.

Judgment $\mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'$ can be read as configuration $\langle c, r, m, K\rangle$ executes in mode μ and terminates with register file r', memory m', kill set K', and during execution produces trace t'.

Evaluation of commands makes use of an additional judgment to evaluate expressions: $\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v$. This judgment means that, given register file r, memory m, and kill set K, expression e evaluates in mode μ to value v. Expression evaluation does not modify the register file, memory, or the kill set.

Figure 2 presents the inference rules for expression evaluation. The rules are straightforward except for DEREF. Integers, locations, and functions are already values.

Rule DEREF evaluates expression e to a memory location l and reads the contents of l. The premise $\delta(l) \in \{N, \mu\} \setminus K$ states that in normal mode (i.e., $\mu = N$), only normal locations can be read; in enclave mode E_i (i.e., $\mu = E_i$), both normal and enclave E_i locations may be read (i.e., $\delta(l) \in \{N, E_i\}$). Locations from a different enclave cannot be read, and if an enclave has been killed ($E_i \in K$), then no locations in that enclave can be read.

Rule ISUNSET returns 1 if the contents of condition cnd is 0 (i.e., if cnd is not set), otherwise it evaluates to 0.

$$\textsc{Int}\ \frac{}{\mu \vdash_\delta \langle n, r, m, K\rangle \Downarrow n} \qquad \textsc{Loc}\ \frac{}{\mu \vdash_\delta \langle l, r, m, K\rangle \Downarrow l}$$

$$\textsc{Var}\ \frac{v = r(x)}{\mu \vdash_\delta \langle x, r, m, K\rangle \Downarrow v} \qquad \textsc{Function}\ \frac{}{\mu \vdash_\delta \langle \lambda^\mu.c, r, m, K\rangle \Downarrow \lambda^\mu.c}$$

$$\textsc{Op}\ \frac{\mu \vdash_\delta \langle e_1, r, m, K\rangle \Downarrow v_1 \quad \mu \vdash_\delta \langle e_2, r, m, K\rangle \Downarrow v_2 \quad v = v_1 \oplus v_2}{\mu \vdash_\delta \langle e_1 \oplus e_2, r, m, K\rangle \Downarrow v} \qquad \textsc{Deref}\ \frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow l \quad m(l) = v \quad \delta(l) \in \{N, \mu\} \setminus K}{\mu \vdash_\delta \langle *e, r, m, K\rangle \Downarrow v}$$

$$\textsc{IsUnset}\ \frac{\mu \vdash_\delta \langle *cnd, r, m, K\rangle \Downarrow n \qquad v = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}}{\mu \vdash_\delta \langle \mathsf{isunset}(cnd), r, m, K\rangle \Downarrow v}$$

Figure 2. Large-step semantics for IMPE expressions

Rule FUNCTION requires that the mode annotation μ on function $\lambda^\mu.c$ equals the mode of the expression evaluation.

Figure 3 shows the inference rules for command execution. Many rules have the premise $\mu \notin K$, which collectively ensure that code in killed enclaves can not be executed. Rules SKIP, ASSIGN, SEQ, IF-ELSE, WHILE-T, and WHILE-F are standard. We write $r[x \mapsto v]$ for the register file that maps x to v but otherwise is the same as r. Similarly, memory $m[l \mapsto v]$ maps l to v but otherwise is the same as m. The command $\mathsf{call}(e)$ evaluates expression e to a function and invokes it. The modes of callee and caller should match. This ensures that function calls cannot be used to amplify privilege, and the only way execution can transition modes is via an $\mathsf{enclave}(i, c)$ command.

Rule UPDATE updates a memory location, and like DEREF, ensures that an enclave's memory locations can be accessed only by code within the enclave. This rule is used only for non-condition locations. Rule SETCND is used to set conditions. Since this is a security relevant action, SETCND produces event $\mathsf{Mem}(m')$ in the trace where m' is the new memory. These events are used in the definition of the semantic security conditions.

Rule DECLASSIFY declassifies expression e and assigns the result to variable x. Operationally, it is similar to ASSIGN but uses predicate $\mathsf{hasNoVars}(e)$ to enforce the syntactic restriction that expression e contains no variables: it may contain expressions built from values and memory locations (including conditions). Declassifications are security relevant events, and so a declassification event $\mathsf{Decl}(e, m)$ is emitted to the trace. Rule OUTPUT evaluates the expression e to

SKIP

$$\frac{\mu \notin K}{\mu \vdash_\delta \langle \mathsf{skip}, r, m, K\rangle \Downarrow r; m; K \triangleright \epsilon}$$

KILL

$$\frac{E_i \notin K}{N \vdash_\delta \langle \mathsf{kill}(i), r, m, K\rangle \Downarrow r; m; K \cup \{E_i\} \triangleright \epsilon}$$

ASSIGN

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v \quad r' = r[x \mapsto v] \quad \mu \notin K}{\mu \vdash_\delta \langle x := e, r, m, K\rangle \Downarrow r'; m; K \triangleright \epsilon}$$

DECLASSIFY

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v \quad \mathsf{hasNoVars}(e) \quad r' = r[x \mapsto v] \quad \mu \notin K}{\mu \vdash_\delta \langle x := \mathsf{declassify}(e), r, m, K\rangle \Downarrow r'; m; K \triangleright \mathsf{Decl}(e, m)}$$

UPDATE

$$\frac{\mu \vdash_\delta \langle e_1, r, m, K\rangle \Downarrow l \quad \mu \vdash_\delta \langle e_2, r, m, K\rangle \Downarrow v \quad \mu \notin K \quad \delta(l) \in \{N, \mu\} \setminus K \quad l \in Loc \setminus Cond \quad m' = m[l \mapsto v]}{\mu \vdash_\delta \langle e_1 \leftarrow e_2, r, m, K\rangle \Downarrow r; m'; K \triangleright \epsilon}$$

SETCND

$$\frac{\delta(cnd) \in \{N, \mu\} \setminus K \quad cnd \in Cond \quad m' = m[cnd \mapsto 1] \quad \mu \notin K}{\mu \vdash_\delta \langle \mathsf{set}(cnd), r, m, K\rangle \Downarrow r; m'; K \triangleright \mathsf{Mem}(m')}$$

OUTPUT

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v \quad \ell \in \{L, H\} \quad \mu \notin K}{\mu \vdash_\delta \langle \mathsf{output}\ e\ \mathsf{to}\ \ell, r, m, K\rangle \Downarrow r; m; K \triangleright \mathsf{Mem}(m) \cdot \mathsf{Out}(\ell, v)}$$

CALL

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow \lambda^\mu.c \quad \mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}{\mu \vdash_\delta \langle \mathsf{call}(e), r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}$$

ENCLAVE

$$\frac{E_i \vdash_\delta \langle c, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}{N \vdash_\delta \langle \mathsf{enclave}(i, c), r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}$$

SEQ

$$\frac{\forall i \in \{1 \ldots n\} \quad \mu \vdash_\delta \langle c_i, r_{i-1}, m_{i-1}, K_{i-1}\rangle \Downarrow r_i; m_i; K_i \triangleright t_i}{\mu \vdash_\delta \langle c_1; \ldots; c_n, r_0, m_0, K_0\rangle \Downarrow r_n; m_n; K_n \triangleright t_1 \cdot \ldots \cdot t_n}$$

IF-ELSE

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v \quad i = \begin{cases} 1 & \text{if } v \neq 0 \\ 2 & \text{otherwise} \end{cases} \quad \mu \vdash_\delta \langle c_i, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}{\mu \vdash_\delta \langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t'}$$

WHILE-T

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow v \quad v \neq 0 \quad \mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow r'; m'; K' \triangleright t' \quad \mu \vdash_\delta \langle \mathsf{while}\ e\ \mathsf{do}\ c, r', m', K'\rangle \Downarrow r''; m''; K'' \triangleright t''}{\mu \vdash_\delta \langle \mathsf{while}\ e\ \mathsf{do}\ c, r, m, K\rangle \Downarrow r''; m''; K'' \triangleright t' \cdot t''}$$

WHILE-F

$$\frac{\mu \vdash_\delta \langle e, r, m, K\rangle \Downarrow 0 \quad \mu \notin K}{\mu \vdash_\delta \langle \mathsf{while}\ e\ \mathsf{do}\ c, r, m, K\rangle \Downarrow r; m; K \triangleright \epsilon}$$

Figure 3. Large-step semantics for select IMPE commands

a value v, outputs v on channel ℓ, and adds events $\mathsf{Mem}(m)$ and $\mathsf{Out}(\ell, v)$ to the trace.

Rule ENCLAVE executes enclave code. Note that enclaves can be entered only from normal mode (i.e., mode μ in the conclusion must be N). This reflects the operation of Intel SGX-like mechanisms: execution of enclave code occurs only by non-enclave code jumping to a well-defined enclave entry point; execution of enclave code ends only by exiting the enclave, not by calling back to non-enclave code; code in one enclave can not directly execute code in another enclave.

Rule KILL tears down enclave E_i and adds it to the kill set. Once an enclave is killed, it is inactive and can no longer be used. Enclaves can be killed only in normal mode.

The following code illustrates how password authentication can be modeled in IMPE.

```
enclave(1, status := *password = *guess);
output status to L
```

The code uses two locations, password and guess, containing the password and user's input respectively. Assume $\delta(\mathsf{password}) = E_1$ and $\delta(\mathsf{guess}) = N$, i.e., password belongs to enclave E_1 and guess is not in an enclave. The program enters enclave E_1, checks if the password matches the guess by dereferencing the corresponding locations, sets variable status to the result, and exits the enclave. Variable status is then output on channel L. Note that dereferencing password would fail if done outside enclave E_1.

3. Attacker Model and Security

In this section we define security for the IMPE language for a variety of attackers. We consider a passive attacker that can only observe outputs on certain channels, an active attacker that can arbitrarily corrupt non-enclave computation, and an active attacker that can, under certain conditions, corrupt computation both outside and inside enclaves.

The definition of security is that at all times, an attacker knows no more than what the attacker is permitted to know. What the attacker is permitted to know is determined by the security policies on information, which conditions are set when, and what declassifications the program performs. We model active attackers by allowing additional transitions in the operational semantics of IMPE. Thus, the definition of security is parameterized on variants of the operational semantics of IMPE.

We assume the only source of confidential information is the initial memory. A *security specification* γ maps locations to policies and indicates the policy to enforce on information in an initial memory. For example, if $\gamma(l) = \ell_1 \overset{cnd}{\nearrow} \ell_2$ and m is the memory from which we start an execution, then we should enforce erasure policy $\ell_1 \overset{cnd}{\nearrow} \ell_2$ on the data in $m(l)$. We say that a security specification γ is *well-formed* if $\forall l \in Loc.\ \gamma(l) \neq \top$ (since security level $\top$ is for information so confidential that it should not be on the machine).

3.1 Attacker Knowledge

We associate an attacker with a security level $\ell \in \mathcal{L}$ and assume the attacker is able to observe outputs on any channel ℓ' such that $\ell' \sqsubseteq \ell$. Given trace t, $\lfloor t \rfloor_\ell$ is the output events that an attacker at level ℓ can observe.

$$\lfloor t \rfloor_\ell = \begin{cases} \mathsf{Out}(\ell', v) \cdot \lfloor t' \rfloor_\ell & \text{if } t = \mathsf{Out}(\ell', v) \cdot t' \text{ and } \ell' \sqsubseteq \ell \\ \lfloor t' \rfloor_\ell & \text{if } t = \mathsf{Out}(\ell', v) \cdot t' \text{ and } \ell' \not\sqsubseteq \ell \\ \lfloor t' \rfloor_\ell & \text{if } t = \alpha \cdot t' \wedge \alpha \neq \mathsf{Out}(\ell', v) \\ \epsilon & \text{otherwise} \end{cases}$$

Given an execution of program c, an attacker at level ℓ observes some portion of the execution (i.e., some subsequence t_{obs} of the trace produced during execution). The knowledge of the attacker is the set of initial memories for which execution of c could produce a trace t such that some subsequence of t looks the same to the attacker as t_{obs}. That is, an attacker's knowledge is the set of initial memories that the attacker believes are possible. Thus, the smaller the attacker's knowledge, the more precise is the attacker's knowledge.

We base our definition of attacker knowledge on that of Askarov et al. [7] by parametrizing it on the large-step semantics. That is, we will instantiate $\Downarrow_{kind}$ with different large-step semantics that represent different attackers. We assume that all initial configurations use a register file r_{init} that maps all variables to zero.

Definition 1 (Attacker knowledge)**.** *Given program c, security level ℓ, large-step semantics $\Downarrow_{kind}$, and trace t_{obs},* attacker knowledge *is defined as:*

$$k_\ell^{\Downarrow_{kind}}(c, t_{obs}) = \{m \mid N \vdash \langle c, r_{init}, m, \emptyset \rangle \Downarrow_{kind} r'; m'; K' \triangleright t \wedge \exists t_0, t_1, t_2.\ t = t_0 \cdot t_1 \cdot t_2 \wedge \lfloor t_{obs} \rfloor_\ell = \lfloor t_1 \rfloor_\ell\}$$

Consider the password authentication example from Section 2.3. Let c be the program, m_0 be the initial memory and $t_{obs} = \mathsf{Mem}(m_0) \cdot \mathsf{Out}(L, 1)$ be the trace produced by the program executed with semantics $\Downarrow$. The knowledge of a passive attacker at security level L is the set of all initial memories such that the contents of locations password and guess are equal. More formally, $k_L^{\Downarrow}(c, t_{obs}) = \{m' \mid m'(\mathsf{password}) = m'(\mathsf{guess})\}$.

3.2 Security

The intuition for knowledge-based security conditions [5, 7] is that an attacker should know only what it is permitted to know. We thus define what an attacker is permitted to know.

We are concerned with attackers that may observe only a portion of a program's execution. Thus, an attacker at level ℓ that starts observing the execution after condition cnd has been set should in general not be able to learn anything about information with erasure policy $\ell_1 \overset{cnd}{\nearrow} \ell_2$ where $\ell_2 \not\sqsubseteq \ell$. However, an attacker is permitted to learn information that has already been declassified, including declassifications that occurred before the attacker started observing the execution.

Permitted knowledge via erasure policies Whether an attacker at level ℓ is permitted to observe information with policy p depends on which conditions have been set. Let $U \subseteq Cond$ be the currently unset conditions. We write $cur(p, U)$ for the security level that should currently be enforced on information with policy p. If p is an erasure policy $\ell_1 \overset{cnd}{\nearrow} \ell_2$, then we should enforce security level ℓ_1 if $cnd \in U$ and enforce ℓ_2 if $cnd \notin U$. Formally:

$$cur(p, U) = \begin{cases} \ell & \text{if } p = \ell \\ \ell_1 & \text{if } p = \ell_1 \overset{cnd}{\nearrow} \ell_2 \text{ and } cnd \in U \\ \ell_2 & \text{if } p = \ell_1 \overset{cnd}{\nearrow} \ell_2 \text{ and } cnd \notin U \end{cases}$$

Based on the current security level to enforce on information, we define equivalence classes of initial memories that an attacker at level ℓ should not be allowed to distinguish. Intuitively, if initial memories m and m' are identical at every location l for which the current security level permits the attacker to learn information (i.e., $cur(\gamma(l), U) \sqsubseteq \ell$), then the attacker should not be allowed to distinguish m and m'.

Definition 2 (Indistinguishable Memories)**.** *Given memory m, security specification γ, unset conditions U, and security level ℓ, we define $ind_\ell(m, \gamma, U)$ as*

$$\{m' \mid \forall l \in Loc.\ cur(\gamma(l), U) \sqsubseteq \ell \implies m(l) = m'(l)\}$$

Given an execution from initial memory m_0 where an attacker at level ℓ starts observing the execution when U are the unset conditions, then the attacker should not learn whether the initial memory for the execution was m_0 or some memory in $ind_\ell(m_0, \gamma, U)$. That is, the attacker's knowledge should be a superset of $ind_\ell(m_0, \gamma, U)$.

Permitted knowledge via escape hatches Declassifications permit an attacker to learn more information. Following Sabelfeld and Myers [30], we use *escape hatches* to characterize what information declassification commands $x := \mathsf{declassify}(e)$ reveal. An escape hatch is a computation over confidential information such that attackers are permitted to learn the result of the computation. In our setting, an escape hatch is an expression e evaluated against the initial memory. Recall that confidential information is input to a program only via the initial memory. Thus, by evaluating escape hatch expression e against the initial memory, e describes a computation over confidential inputs that is permitted to be declassified.

We connect declassification events $\mathsf{Decl}(e, m)$ (where m is the current memory at the time of declassification, and expression e contains only operations over constants and memory locations) to escape hatches by requiring that the evaluation of e using m produces the same value as the evaluation of e using the initial memory. If so, the attacker is permitted to learn the result of e, otherwise we do not allow the declassification event to release any information. We capture this in the definition of escape-hatch indistinguishability below.

Definition 3 (Escape-hatch indistinguishability)**.** *Given initial memory m_0, current memory m, semantics $\Downarrow_{kind}$ and escape hatch e, we define $Esc^{\Downarrow_{kind}}(m_0, m, e)$ as*

$$\{m' \mid \exists \mu.\ (\mu \vdash_\delta \langle e, r_{init}, m_0, \emptyset\rangle \Downarrow_{kind} v \ \wedge \ \mu \vdash_\delta \langle e, r_{init}, m, \emptyset\rangle \Downarrow_{kind} v) \implies \mu \vdash_\delta \langle e, r_{init}, m', \emptyset\rangle \Downarrow_{kind} v\}$$

Given semantics $\Downarrow_{kind}$, declassification event $\mathsf{Decl}(e, m)$, and initial memory m_0, $Esc^{\Downarrow_{kind}}(m_0, m, e)$ is equal to the set of all initial memories if expression e evaluates to different values in m and m_0 (i.e., the attacker should not learn any information from the declassification), and otherwise is equal to all initial memories m' such that e evaluates to the same value in m' as it does in m_0 (i.e., the attacker is permitted to learn the result of evaluating e).

Security definition We define $\lfloor t \rfloor_{mem} = \{m \mid \mathsf{Mem}(m) \in t\}$ to be the set of memory events that occur in trace t and $\lfloor t \rfloor_{esc} = \{(e, m) \mid \mathsf{Decl}(e, m) \in t\}$ to be the set of tuples corresponding to the declassification events in trace t.

Suppose we have an execution from initial memory m_0 with specification γ that produces trace $t \cdot t_{obs} \cdot t'$, where an attacker at level ℓ observes t_{obs}. Then the attacker is permitted to learn any information that a memory $m' \in \lfloor t_{obs} \rfloor_{mem}$ permits. That is, the intersection of the sets $ind_\ell(m_0, \gamma, \{cnd \mid m'(cnd) = 0\})$ for $m' \in \lfloor t_{obs} \rfloor_{mem}$ describes what information the attacker is permitted to know based on the current security levels of information.

Moreover, the attacker is allowed to learn declassified information. The intersection of sets $Esc^{\Downarrow_{kind}}(m_0, m', e')$ for $(e', m') \in \lfloor t \cdot t_{obs} \rfloor_{esc}$ describes what information the attacker is permitted to know based on declassifications that occurred before or during the attacker observation.

A program is secure if the attacker's knowledge is indeed no more precise than the information the attacker is permitted to know. Definition 4 captures this intuition.

Definition 4 (Security)**.** *Program c is* secure *at security level ℓ for security specification γ and large-step semantics $\Downarrow_{kind}$ if for all initial memories m_0 and all executions*

$$N \vdash_\delta \langle c, r_{init}, m_0, \emptyset\rangle \Downarrow_{kind} r; m; K \triangleright t \cdot t_{obs} \cdot t'$$

where $t_{obs} = \mathsf{Mem}(m'') \cdot t''$ for some memory m'' and trace t'', we have

$$k_\ell^{\Downarrow_{kind}}(c, t_{obs}) \supseteq \left(\bigcap_{m' \in \lfloor t_{obs} \rfloor_{mem}} ind_\ell(m_0, \gamma, \{cnd \mid m'(cnd) = 0\}) \ \cap \bigcap_{(e', m') \in \lfloor t \cdot t_{obs} \rfloor_{esc}} Esc^{\Downarrow_{kind}}(m_0, m', e') \right)$$

Note that this definition is termination- and progress-insensitive [6]. We can modify the definition to be termination- and progress-sensitive, but this results in a more complicated definition that does not provide additional insight into the issues explored in this paper. We thus refrain from doing so.

Note that the definition quantifies over all possible observations t_{obs}. The definition requires that the first event in the observed trace t_{obs} is a memory event to ensure we know the current security level to enforce on information as at the start of the attacker's observation. This is without loss of generality, since every output event is immediately preceded by a memory event (see rule OUTPUT in Figure 3).

For example, let c be the password authentication program modified to set condition end on enclave exit.

$$\mathsf{enclave}(1, \mathsf{status} := *\mathsf{password} = *\mathsf{guess});\ \mathsf{set}(\mathsf{end});\ \mathsf{output\ status\ to}\ L$$

The program is insecure for the specification γ, where $\gamma(\mathsf{guess}) = L \overset{\mathsf{end}}{\nearrow} \top$ and $\gamma(\mathsf{password}) = H$. Intuitively, for initial memory m_0 and $t_{obs} = \mathsf{Mem}(m_0[\mathsf{end} \mapsto 1]) \cdot \mathsf{Out}(L, 1)$ produced by execution with semantics $\Downarrow$, then the lower bound $ind_L(m_0, \gamma, \emptyset)$ on the knowledge of an attacker at security level L is the set of all memories. However, the attacker learns that password and guess are equal.

Suppose we now modify the program to include declassification:

$$\mathsf{enclave}(1, \mathsf{status} := \mathsf{declassify}(*\mathsf{password} = *\mathsf{guess})); \ldots.$$

The declassification event induces a new lower bound: $\{m' \mid m'(\mathsf{password}) = m'(\mathsf{guess})\}$ which is same as the attacker's knowledge. The program is now secure for an attacker at security level L.

N-CHAOS

$$\frac{c \overset{enc}{\simeq} c' \qquad N \vdash_\delta \langle c', r, m, K\rangle \Downarrow_{N\text{-}chaos} r'; m'; K' \triangleright t'}{N \vdash_\delta \langle c, r, m, K\rangle \Downarrow_{N\text{-}chaos} r'; m'; K' \triangleright \mathsf{Mem}(m) \cdot \mathsf{A}(c \overset{enc}{\simeq} c') \cdot t'}$$

Figure 4. Additional inference rule for $\Downarrow_{N\text{-}chaos}$

E$_I$-CHAOS

$$\frac{I \subseteq K \qquad N \vdash_\delta \langle c', r, m, K\rangle \Downarrow_{E_I\text{-}chaos} r'; m'; K' \triangleright t'}{N \vdash_\delta \langle c, r, m, K\rangle \Downarrow_{E_I\text{-}chaos} r'; m'; K' \triangleright \mathsf{Mem}(m) \cdot \mathsf{A}(c') \cdot t'}$$

Figure 5. Additional inference rule for $\Downarrow_{E_I\text{-}chaos}$

3.3 Attackers

A *passive attacker* simply observes the execution of program and attempts to learn information about confidential input. By contrast, an *active attacker* can modify or influence the execution of a program. Active attackers represent many malicious behaviors, including attacks that can modify execution arbitrarily (e.g., by gaining control of the program counter or overwriting code) or modify some set of memory locations (e.g., by buffer overflows or by providing malicious input to a program).

We consider three attackers: (1) A *passive attacker* that can only observe output on the L channel; (2) A *non-enclave active attacker* that can observe output on the L channel and arbitrarily modify non-enclave code; and (3) An *enclave active attacker* that can observe output on the L and H channels, and can arbitrarily modify (enclave and non-enclave) code but only under certain conditions.

We use different operational semantics to represent the different attackers. The passive attacker corresponds to the semantics $\Downarrow$ (Figure 3). That is, programs execute as written, and the attacker passively observes output. We define two new semantics to capture the abilities of the active attackers.

Non-enclave active attacker Relation $\Downarrow_{N\text{-}chaos}$ allows the attacker to arbitrarily change non-enclave code. Inference rules for judgment $\mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow_{N\text{-}chaos} r'; m'; K' \triangleright t$ include all rules from Figure 3 (appropriately adapted) and the rule in Figure 4. This new rule allows command c to change to command c', so long as both commands have the same enclave code, expressed by relation $c \overset{enc}{\simeq} c'$ (defined in Appendix A). This corresponds to an attacker that can exploit a vulnerability in non-enclave code but is unable to corrupt code within enclaves. Since modifying the program is a security relevant action, an event $\mathsf{A}(c \overset{enc}{\simeq} c')$ is emitted to the trace (and we modify $\lfloor t \rfloor_\ell$ to include events of the form $\mathsf{A}(c \overset{enc}{\simeq} c')$).

If a program is secure for $\Downarrow_{N\text{-}chaos}$ then it is secure for $\Downarrow$. The converse does not necessarily hold. For example, consider the following program, where $\delta(\mathsf{hi}) = E_1$ and $\gamma(\mathsf{hi}) = H$.

$$c \;\equiv\; \mathsf{enclave}(1, \mathsf{x} := *\mathsf{hi}); \mathsf{output}\ 1\ \mathsf{to}\ L$$

The program is secure at level L for specification γ and semantics $\Downarrow$ but is insecure for semantics $\Downarrow_{N\text{-}chaos}$. Suppose the active attacker modifies the program to $c' \equiv \mathsf{enclave}(1, \mathsf{x} := *\mathsf{hi}); \mathsf{output}\ \mathsf{x}\ \mathsf{to}\ L$. Note that $c \overset{enc}{\simeq} c'$, since the code in enclaves for both c and c' is the same: $\mathsf{enclave}(1, \mathsf{x} := *\mathsf{hi})$. Suppose we execute c' with an initial memory that maps hi to 42. The knowledge of an attacker observing output on channel L is $\{m' \mid m'(\mathsf{hi}) = 42\}$. However the permitted lower bound on attacker's knowledge is the set of all initial memories (i.e., the attacker is not permitted to learn anything about the confidential data). So the program is not secure at level L for γ and $\Downarrow_{N\text{-}chaos}$.

Enclave active attacker Given a set of enclaves $I \subseteq \{E_1, E_2, \dots\}$, relation $\Downarrow_{E_I\text{-}chaos}$ allows the attacker to arbitrarily change (enclave and non-enclave) code but only after all enclaves in I are killed. This corresponds to a setting where enclaves and non-enclave code all have exploitable vulnerabilities but the attacker does not immediately exploit these vulnerabilities.

Inference rules for judgment $\mu \vdash_\delta \langle c, r, m, K\rangle \Downarrow_{E_I\text{-}chaos} r'; m'; K' \triangleright t$ include all inference rules from Figure 3 (appropriately adapted) and the rule in Figure 5. This new rule allows command c to change to an arbitrary command c' provided all enclaves in I are killed ($I \subseteq K$). Event $\mathsf{A}(c')$ is emitted to the trace (and we modify $\lfloor t \rfloor_\ell$ to include events of the form $\mathsf{A}(c')$).

Consider the following program that stores credit card information in an enclave, where $\gamma(\mathsf{ccard}) = H \overset{\mathsf{end}}{\nearrow} \top$ (and $\gamma(l) = L$ for all other locations l) and $\delta(\mathsf{ccard}) = E_1$.

$$\mathsf{enclave}(1, \mathsf{output}\ *\mathsf{ccard}\ \mathsf{to}\ H; \mathsf{set}(\mathsf{end})); \mathsf{kill}(1)$$

The program outputs the contents of ccard to the H channel and sets condition cnd in enclave E_1. It then exits the enclave E_1 and kills it. The program is secure at security level H for specification γ and semantics $\Downarrow_{E_I\text{-}chaos}$, where $I = \{E_1\}$. This means that if an enclave active attacker modifies the program after $\mathsf{kill}(1)$ has been executed, it is unable to learn anything about the contents of ccard. Note that when $I = \emptyset$, the program is not secure for any subset of locations.

4. IMPE Type System

We introduce a security-type system [38] for IMPE that guarantees security, i.e., a well-typed program is secure against all the attackers described in Section 3.3.

Figure 6 shows the syntax of types. Base types σ include integers, conditions, references, and functions. Recall that conditions are a subset of locations. We use type cond^μ for conditions (i.e., values in the set $Cond$). A reference type $\tau^\mu\ \mathsf{ref}^{rt}$ is a pointer to a memory location with contents of type τ. Both condition types and reference types are annotated with mode μ, indicating in which enclave, if any, the memory location resides. Reference types also have a mutability annotation $rt \in \{\mathsf{mut}, \mathsf{immut}\}$, indicating whether the reference is mutable or immutable. We use immutable refer-

$$\sigma ::= \mathsf{int} \mid \mathsf{cond}^{\mu} \mid \tau^{\mu}\ \mathsf{ref}^{rt} \mid \Gamma^{-}, K^{-}, U \xrightarrow{p,\mu} \Gamma^{+}, K^{+}$$
$$\tau ::= \sigma_p$$
$$\mu \in Mode = \{N, E_1, E_2, \dots\} \qquad p, pc \in P$$
$$rt \in \{\mathsf{mut}, \mathsf{immut}\}$$

Figure 6. IMPE types

ences to ensure that declassified expressions are indeed escape hatches, i.e., that declassified expressions are not modified prior to declassification. All conditions are mutable. We explain function types $\Gamma^{-}, K^{-}, U \xrightarrow{p,\mu} \Gamma^{+}, K^{+}$ after explaining the type judgment.

A security type $\tau = \sigma_p$ is a base type σ annotated with a security policy p. Intuitively, data with type σ_p should have security policy p or a more restrictive policy enforced on it.

A type environment Γ maps variables to security types, and non-condition locations to pairs (τ, rt) of a security type and immutability annotation, where τ is the type of the location's contents, and rt describes the location's immutability. For simplicity, we assume that whether a condition is set is public information, and so for any $cnd \in Cond$, the type of cnd is cond^{μ}_{L} for some mode μ where $\delta(cnd) = \mu$. Thus, we exclude $Cond$ from the domain of Γ.

A type environment is well-typed for δ if all locations containing confidential information belong to some enclave. Since security level $\top$ is meant to indicate information that is too confidential to be stored on the machine, we also require that well-typed environments do not map any variable or location to a type $\sigma_{\top}$. The following definition formally states the well-typedness of environment Γ for δ.

Definition 5 (Well-Typed Environment)**.** *A type environment Γ is well-typed for δ, denoted as $\vdash_{\delta} \Gamma\ \mathsf{ok}$, if*

$$\forall l \in Loc \setminus Cond.\ \Gamma(l) = (\sigma_p, rt) \wedge p \not\leq L \implies \delta(l) \neq N \wedge p \neq \top$$

and

$$\forall x \in Vars.\ \Gamma(x) = \sigma_p \implies p \neq \top$$

The IMPE type system is flow-sensitive in that the type of variables may differ at different program points.[1] Also, our type system tracks the set of killed enclaves to ensure that no code or data inside a killed enclave is accessed. To ensure that erasure policies are correctly enforced, our type system tracks the set of conditions that are definitely unset. The typing judgment for commands has the form

$$pc, \mu, \Gamma, K, U \vdash_{\delta} c : \Gamma', K'$$

where:

- Γ and Γ' are, respectively, the type environments immediately before and after execution of command c;
- K and K' are, respectively, the set of killed enclaves immediately before and after execution of c;
- U is the set of conditions that are known to be not set immediately before the execution of c;
- μ indicates whether c executes in normal mode ($\mu = N$) or in an enclave ($\mu = E_i$);
- pc is a policy representing an upper bound on the information that influences the decision to execute c, and is also a lower bound on the side-effects of c. This *program counter policy* [29, 38] is used to prevent *implicit flows* [14], i.e., information flows via the control decisions of a program.
- δ is a function which indicates for each memory location which enclave, if any, it belongs to.

The type judgment for expressions is $\mu, \Gamma \vdash_{\delta} e : \tau$, meaning that in mode μ under type environment Γ, expression e has type τ.

A function type $\Gamma^{-}, K^{-}, U \xrightarrow{p,\mu} \Gamma^{+}, K^{+}$ indicates the type environment Γ^{-} that must hold before the function is invoked, and the type environment Γ^{+} that will hold immediately after function invocation. These environments may be partial functions, since the function may use only a subset of variables. Well-typedness of functions will ensure that $\mathrm{dom}(\Gamma^{-}) \subseteq \mathrm{dom}(\Gamma^{+})$. Kill set K^{-} is the set of killed enclaves the function expects at invocation, and K^{+} is the set of killed enclaves after function invocation. Set U is the set of conditions that the function assumes are unset upon function invocation. Mode μ is the mode in which the function was defined, and policy p is a lower bound on the side-effects of the function.

We define subtyping on security types based on the relative restrictiveness of security policies. Given policies p and q, we say that q is at least as restrictive as p, written $p \leq q$, if policy q imposes at least as many restrictions on the use of data as policy p. The relation $\leq$ on policies forms a lattice. We write $\sqcup$ for the join operation. We overload the symbol $\leq$ and write $\sigma_1 \leq \sigma_2$ when base type σ_1 is a subtype of base type σ_2, and write $\tau_1 \leq \tau_2$ when security type τ_1 is a subtype of security type τ_2. We lift subtyping to type environments and define $\Gamma_1 \leq \Gamma_2$ if and only if $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ and $\forall y \in \mathrm{dom}(\Gamma_1).\ \Gamma_1(y) \leq \Gamma_2(y)$. Function types are contravariant in the pre-environment, and the side-effect bound, covariant in the post-environment, and invariant otherwise. Inference rules for the subtyping ($\leq$) relation are presented in Appendix B.

Figure 7 shows typing rules for expressions. As is standard in security-type systems, constants (including integers, conditions, references, and function definitions) are given policy L, the most permissive security policy.

Dereferencing an expression may reveal information about both which location is dereferenced and the contents of that location. Thus in rule T-DEREF the result of a $*e$ ex-

[1] The type system is not flow-sensitive for locations. Although this could be achieved using alias information, for simplicity we assume that the types of locations are fixed throughout the program.

T-INT

$$\frac{}{\mu, \Gamma \vdash_\delta n : \mathsf{int}_L}$$

T-CND

$$\frac{\mu' = \delta(cnd) \qquad cnd \in Cond}{\mu, \Gamma \vdash_\delta cnd : \mathsf{cond}^{\mu'}{}_L}$$

T-VAR

$$\frac{\Gamma(x) = \sigma_p}{\mu, \Gamma \vdash_\delta x : \sigma_p}$$

T-LOC

$$\frac{\Gamma(l) = (\sigma_p, rt) \qquad \mu' = \delta(l) \qquad l \in Loc \setminus Cond}{\mu, \Gamma \vdash_\delta l : (\sigma_p^{\mu'}\ \mathsf{ref}^{rt})_L}$$

T-DEREF

$$\frac{\mu, \Gamma \vdash_\delta e : \sigma_p^{\mu'}\ \mathsf{ref}^{rt}{}_q \qquad \mu' \neq N \implies \mu = \mu'}{\mu, \Gamma \vdash_\delta *e : \sigma_{p \sqcup q}}$$

T-ISUNSET

$$\frac{\mu' = \delta(cnd) \qquad cnd \in Cond \qquad \mu' \neq N \implies \mu = \mu'}{\mu, \Gamma \vdash_\delta \mathsf{isunset}(cnd) : \mathsf{int}_L}$$

T-FUNCTION

$$\frac{p, \mu, \Gamma^-, K^-, U \vdash_\delta c : \Gamma^+, K^+}{\mu, \Gamma \vdash_\delta \lambda^\mu.c : (\Gamma^-, K^-, U \xrightarrow{p,\mu} \Gamma^+, K^+)_L}$$

T-OP

$$\frac{\mu, \Gamma \vdash_\delta e_1 : \mathsf{int}_p \qquad \mu, \Gamma \vdash_\delta e_2 : \mathsf{int}_q}{\mu, \Gamma \vdash_\delta e_1 \oplus e_2 : \mathsf{int}_{p \sqcup q}}$$

Figure 7. IMPE typing rules for expressions

pression has a security policy that is at least as restrictive as the policy on the reference and the contents of the reference. The premise $\mu' \neq N \implies \mu = \mu'$ (in both T-DEREF and T-ISUNSET) requires that locations in enclaves can be accessed only by code in the same enclave.

Most of the commands follow the standard security typing rules for an imperative language (including subsumption). The rules further ensure that killed enclaves are never accessed (premise $\mu \notin K$ in many rules) and that enclave locations are accessed only by code in the appropriate enclave, that only public information (i.e., with security policy L) can be accessed outside of enclaves (premise $p \not\leq L \implies \mu' \neq N$ in many rules), and that the program does not store information at security level $\top$ (premise $p \neq \top$ in many rules). To ensure that kill sets are tracked precisely, we require that both branches of conditionals kill the same enclaves, and that the body of loops kill no enclaves. We also require functions that expect non-empty U to in an enclave. This prevents non-enclave attackers from violating the assumption on U when invoking a function.

Figure 8 presents typing rules for commands.[2] Rules T-SKIP, T-ASSIGN, T-SUB, T-SEQ, T-IF-ELSE, T-WHILE are mostly standard. Rule T-DECLASSIFY ensures that a declassification uses only immutable locations and has no variables (so that the value declassified is the same as the valuation of the expression in the initial memory, and so the expression is an escape hatch). This is enforced by premises $\mathsf{hasNoVars}(e)$ and $\mathsf{allLocImmutable}(e)$. The latter is formally defined as $\forall l \in e.\ l \notin Cond \wedge (\Gamma(l) = (\sigma_p, rt) \implies rt = \mathsf{immut})$. To ensure that the decision to declassify information does not reveal information, we require that the program counter policy for a declassification is L. Note that the result of the declassification has policy L (i.e., variable x maps to σ_L).

Rule T-OUTPUT ensures that the current security level enforced on both a value output to a channel, and the decision to perform the output, is permitted to be output ($cur(p, U) \sqcup cur(pc, U) \sqsubseteq \ell$).

Rule T-IF-ISUNSET is similar to T-IF-ELSE but is used when the branch condition tests whether a condition is unset. This allows the true branch to be type-checked under the assumption that condition cnd is unset, which improves precision, of, for example, output commands.

Rule T-ENCLAVE ensures that command $\mathsf{enclave}(i, c)$ can be executed only in non-enclave mode $\mu = N$, and type checks c in mode E_i with an empty set of conditions that are assumed to be unset. This is to ensure that an attacker that can control non-enclave execution is unable to violate an assumption made by enclave code. Also, premise $\mathsf{isVarLowContext}(\Gamma')$ ensures that at the end of the enclave code, all variables contain only information with policy L This ensures that at the end of execution of enclave code, there is no confidential information remaining in variables that could be leaked to an attacker.

Rule T-KILL requires that enclaves can only be killed by non-enclave code ($\mu = N$). This reflects the operation of Intel's SGX enclaves. Since killing an enclave may be detectable by a non-enclave attacker, we ensure that the decision to kill an enclave relies on only non-confidential information ($pc = L$). Premise $E_i \notin K$ ensures that an enclave is not killed more than once.

Rule T-UPDATE is mostly standard, but like T-DEREF and T-ISUNSET, requires that locations in enclaves can be accessed only by code in the same enclave.

Rule T-SETCND checks that condition cnd can be set only if it is not currently assumed to be unset, i.e., $cnd \notin U$.

Rule T-CALL ensures that the preconditions for calling the function are satisfied, namely that the kill set and unset conditions assumed by the function is equal to the current kill set and unset conditions, and that the assumptions of the function's type environment are satisfied ($\forall y \in \mathrm{dom}(\Gamma^-), \Gamma(y) \leq \Gamma^-(y)$). The program counter policy pc and the information revealed by which function to invoke (q) must be no more restrictive than p, the lower bound on the function's side effects. The premise $U \neq \emptyset \implies \mu \neq N$ prevents a non-enclave active attacker from directly invoking a function and violating the assumption on set U. The type environment after the function call respects the function's post-environment: $\forall y \in \mathrm{dom}(\Gamma^+), \Gamma^+(y) \leq \Gamma_{out}(y)$. Since

[2] For presentation purposes, these rules are non-algorithmic. They can be easily adapted to enable syntax-directed type-checking algorithms.

T-SKIP

$$\frac{\mu \notin K}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{skip} : \Gamma, K}$$

T-KILL

$$\frac{E_i \notin K \qquad pc = L \qquad \mu = N}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{kill}(i) : \Gamma, K \cup \{E_i\}}$$

T-ASSIGN

$$\frac{\mu, \Gamma \vdash_\delta e : \sigma_p \qquad pc \sqcup p \neq \top \qquad (pc \sqcup p) \not\leq L \implies \mu \neq N \qquad \mu \notin K}{pc, \mu, \Gamma, K, U \vdash_\delta x := e : \Gamma[x \mapsto \sigma_{pc \sqcup p}], K}$$

T-DECLASSIFY

$$\frac{\mu, \Gamma \vdash_\delta e : \sigma_p \qquad \mu \notin K \qquad p \neq \top \qquad pc = L \qquad \mathsf{hasNoVars}(e) \qquad \mathsf{allLocImmutable}(e)}{pc, \mu, \Gamma, K, U \vdash_\delta x := \mathsf{declassify}(e) : \Gamma[x \mapsto \sigma_L], K}$$

T-OUTPUT

$$\frac{\mu, \Gamma \vdash_\delta e : \sigma_p \qquad \mu \notin K \qquad p \neq \top \qquad cur(p, U) \sqcup cur(pc, U) \sqsubseteq \ell}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{output}\ e\ \mathsf{to}\ \ell : \Gamma, K}$$

T-UPDATE

$$\frac{\mu, \Gamma \vdash_\delta e_1 : (\sigma_p^{\mu'}\ \mathsf{ref}^{rt})_q \qquad \mu, \Gamma \vdash_\delta e_2 : \sigma_{p'} \qquad p' \sqcup q \sqcup pc \leq p \qquad \mu' \neq N \implies \mu = \mu' \qquad \mu \notin K \qquad rt = \mathsf{mut} \qquad p, p', q \neq \top}{pc, \mu, \Gamma, K, U \vdash_\delta e_1 \leftarrow e_2 : \Gamma, K}$$

T-SEQ

$$\frac{\forall i \in \{1 \ldots n\}.\ pc, \mu, \Gamma_{i-1}, K_{i-1}, U \vdash_\delta c_i : \Gamma_i, K_i}{pc, \mu, \Gamma_0, K_0, U \vdash_\delta c_1; \ldots; c_n : \Gamma_n, K_n}$$

T-SETCND

$$\frac{\mu' = \delta(cnd) \qquad pc = L \qquad cnd \in Cond \setminus U \qquad \mu' \neq N \implies \mu = \mu' \qquad \mu \notin K}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{set}(cnd) : \Gamma, K}$$

T-IF-ISUNSET

$$\frac{\mu, \Gamma \vdash_\delta \mathsf{isunset}(cnd) : \mathsf{int}_L \qquad pc, \mu, \Gamma, K, U \cup \{cnd\} \vdash_\delta c_1 : \Gamma', K' \qquad pc, \mu, \Gamma, K, U \vdash_\delta c_2 : \Gamma', K'}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{if}\ \mathsf{isunset}(cnd)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \Gamma', K'}$$

T-IF-ELSE

$$\frac{pc', \mu, \Gamma, K, U \vdash_\delta c_1 : \Gamma', K' \qquad \mu, \Gamma \vdash_\delta e : \mathsf{int}_p \qquad pc \sqcup p \leq pc' \qquad pc', \mu, \Gamma, K, U \vdash_\delta c_2 : \Gamma', K' \qquad p \not\leq L \implies \mu \neq N \qquad p \neq \top}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \Gamma', K'}$$

T-ENCLAVE

$$\frac{pc, E_i, \Gamma, K, \emptyset \vdash_\delta c : \Gamma', K' \qquad \mathsf{isVarLowContext}(\Gamma') \qquad \mu = N}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{enclave}(i, c) : \Gamma', K'}$$

T-WHILE

$$\frac{\mu, \Gamma \vdash_\delta e : \mathsf{int}_p \qquad pc', \mu, \Gamma, K, U \vdash_\delta c : \Gamma, K \qquad pc \sqcup p \leq pc' \qquad p \not\leq L \implies \mu \neq N \qquad p \neq \top}{pc, \mu, \Gamma, K, U \vdash_\delta \mathsf{while}\ e\ \mathsf{do}\ c : \Gamma, K}$$

T-SUB

$$\frac{pc_1, \mu, \Gamma_1, K, U \vdash_\delta c : \Gamma_1', K' \qquad pc_2 \leq pc_1 \qquad \Gamma_2 \leq \Gamma_1 \qquad \Gamma_1' \leq \Gamma_2' \qquad \vdash_\delta \Gamma_2\ \mathsf{ok} \qquad \vdash_\delta \Gamma_2'\ \mathsf{ok}}{pc_2, \mu, \Gamma_2, K, U \vdash_\delta c : \Gamma_2', K'}$$

T-CALL

$$\frac{\mu, \Gamma \vdash_\delta e : (\Gamma^-, K^-, U \xrightarrow{p,\mu} \Gamma^+, K^+)_q \qquad pc \sqcup q \leq p \qquad \forall y \in \mathrm{dom}(\Gamma^-), \Gamma(y) \leq \Gamma^-(y) \qquad \forall y \in \mathrm{dom}(\Gamma^+).\ \Gamma^+(y) \leq \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^+).\ \Gamma(y) = \Gamma_{out}(y) \qquad q \neq \top \qquad U \neq \emptyset \implies \mu \neq N}{pc, \mu, \Gamma, K^-, U \vdash_\delta \mathsf{call}(e) : \Gamma_{out}, K^+}$$

Figure 8. IMPE typing rules for commands

Γ^- and Γ^+ are partial, we require that the types of variables not in $\mathrm{dom}(\Gamma^+)$ (which is a superset of $\mathrm{dom}(\Gamma^-)$) remain unchanged: $\forall y \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^+).\ \Gamma(y) = \Gamma_{out}(y)$. After the function invocation, the kill set is K^+.

Type Soundness Program execution starts with a known initial register file (r_{init}) that maps all variables to constant zero. We say that type environment Γ corresponds to security specification γ if policies on locations agree with γ and Γ maps all variables to int_L (since r_{init} maps every variable to zero). Formally, Γ corresponds to security specification γ if $\forall l \in \mathrm{dom}(\gamma).\ \gamma(l) = p \implies \Gamma(l) = \sigma_p$ and $\forall x \in \mathit{Vars}.\ \Gamma(x) = \mathsf{int}_L$.

Given a type environment Γ that corresponds to a well-formed security specification γ and is also well-typed for δ, the type system is sound. That is, a well-typed program is secure against all the attackers described in Section 3.3.

Theorem 1. *Let γ be a well-formed security specification and Γ be a type environment that corresponds to γ and is well-typed for δ. If $L, N, \Gamma, \emptyset, \emptyset \vdash_\delta c : \Gamma', K'$ then:*

- *c is secure at security level L for specification γ and semantics $\Downarrow$; and*
- *c is secure at security level L for specification γ and semantics $\Downarrow_{N\text{-}chaos}$; and*
- *For all $I \subseteq \{E_1, E_2, \ldots\}$, define*

$$\gamma'(l) = \begin{cases} \gamma(l) & \text{if } \delta(l) \in I \\ L & \text{otherwise} \end{cases}$$

Command c is secure at security level H for specification γ' and semantics $\Downarrow_{E_I\text{-}chaos}$.

$$
\begin{aligned}
e &::= n \mid x \mid e_1 \oplus e_2 \mid l \mid *e \mid \mathsf{isunset}(cnd) \mid \lambda.c \\
v &::= \lambda.c \mid n \mid l \\
c &::= \mathsf{skip} \mid x := e \mid x := \mathsf{declassify}(e) \mid e_1 \leftarrow e_2 \\
&\quad \mid \mathsf{output}\ e\ \mathsf{to}\ \ell \mid \mathsf{call}(e) \mid \mathsf{set}(cnd) \mid c_1; \ldots; c_n \\
&\quad \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
l &\in Loc \qquad Cond \subset Loc \qquad cnd \in Cond \\
\underline{\sigma} &::= \underline{\mathsf{int}} \mid \underline{\mathsf{cond}} \mid \underline{\tau}\ \mathsf{ref}^{rt} \mid \mathcal{G}^-, U \xrightarrow{p} \mathcal{G}^+ \\
\underline{\tau} &::= \underline{\sigma}_p \\
rt &\in \{\mathsf{mut}, \mathsf{immut}\}
\end{aligned}
$$

Figure 9. IMPS syntax and types

Note that for an enclave active attacker that can attack enclaves in set I only after those enclaves have been killed, Theorem 1 states that command c is secure for security specification γ' derived from γ. Specification γ' is the same as γ for all locations placed in enclaves in I, but for all other locations does not enforce any security restrictions (i.e., $\gamma'(l) = L$ if $\delta(l) \notin I$). That is, we can protect information placed in enclaves in I against an enclave active attacker characterized by semantics $\Downarrow_{E_I\text{-}chaos}$, but can not provide guarantees about information placed in other enclaves.

5. IMPS: a Non-Enclave Calculus

Enclaves are a powerful mechanism, but management of enclaves may be error prone and distract the programmer from implementing correct functionality. We present a language IMPS that is similar to IMPE but removes all enclave-related commands and abstractions and thus allows the programmer to focus on functionality and high-level security requirements. In Section 6 we translate from IMPS to IMPE, automatically inferring appropriate enclaves.

The syntax for IMPS (Figure 9) is similar to IMPE, except that functions are not annotated with mode and commands $\mathsf{enclave}(i, c)$ and $\mathsf{kill}(i)$ are removed. An IMPS configuration is a triple $\langle c, r, m\rangle$ where r and m are a register file and memory, respectively, as defined in Section 2.3.

The large-step semantic judgment for IMPS has the form $\langle c, r, m\rangle \Downarrow_s r'; m' \triangleright t'$, meaning configuration $\langle c, r, m\rangle$ executes and terminates with register file r' and memory m' and during execution produces trace t'. The judgment for expression evaluation is $\langle e, r, m\rangle \Downarrow_s v$ and can be read as given register file r and memory m, expression e evaluates to value v. Inference rules for these judgments are straightforward, and similar to those of IMPE, although without any restrictions based on modes.

Types in IMPS (Figure 9) are similar to those of IMPE. We underline types and type metavariables to distinguish them from IMPE types. Unlike IMPE, conditions and references do not have mode annotations, and function types have neither mode nor kill set annotations. We use $\mathcal{G}$ to denote type environments in IMPS. A type environment $\mathcal{G}$ is well-typed if it does not map any location to type $\underline{\sigma}_\top$. Similar to Γ, we say that type environment $\mathcal{G}$ corresponds to security specification γ, if $\mathcal{G}$ maps all variables to $\underline{\mathsf{int}}_L$ and policies on locations agree with γ.

The type system is a simplified version of the IMPE type system. Judgment $pc, \mathcal{G}, U \vdash c\colon \mathcal{G}'$ means that command c is well-typed, where $\mathcal{G}$ and $\mathcal{G}'$ are the type environments immediately before and after execution of c, program counter policy pc is an upper bound on the information that influences the decision to execute c, and set U are conditions that are definitely unset.

Judgment $\mathcal{G} \vdash e\colon \underline{\sigma}_p$ means expression e has type $\underline{\sigma}_p$ under type environment $\mathcal{G}$. All typing rules are straightforward adaptations of the IMPE rules and are given in the accompanying technical report [20].

6. Translation

We provide a translation of IMPS programs to IMPE that automatically places code and locations into enclaves. Our translation is constraint-based: a type-directed algorithm produces a set of constraints, and any solution to the constraints provides a well-typed IMPE program. We consider various criteria for choosing one translation over another.

6.1 Constraint-based Translation

The constraints place restrictions on where locations may be placed (i.e., on the function δ), on kill sets, on mode annotations, and on which commands may be placed inside an enclave.

The constraints ensure that any location that contains confidential information (i.e., with a policy other than L) is placed in an enclave, and that subsequent use of these locations is consistent (i.e., accessed only by code in the same enclave). The constraints also ensure that confidential data cannot be accessed in non-enclave mode, and also that enclaves are killed appropriately (i.e., no enclave is accessed after it is killed, and enclaves are killed at most once).

The translation judgment for commands has the form

$$pc, \mathcal{G}, U, c, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, c', \Gamma', K'$$

We ensure that if $pc, \mathcal{G}, U \vdash c\colon \mathcal{G}'$ for IMPS command c, then c' is the translated IMPE command such that, provided the constraints are satisfied, $pc, \mu, \Gamma, K, U \vdash_\delta c : \Gamma', K'$.

Instead of the translation judgment explicitly producing a set of constraints, for brevity we present inference rules for the judgment such that constraints are implied by premises that restrict modes, mode annotations, kill sets, etc.

The translation judgment for expressions has the form $\mathcal{G}, e, \underline{\sigma}_p \rightsquigarrow \mu, \Gamma, \delta, e', \sigma_p$ where e is an IMPS expression such that $\mathcal{G} \vdash e\colon \underline{\sigma}_p$ holds and e' is the translated IMPE expression such that, provided the constraints are satisfied, $\mu, \Gamma \vdash_\delta e' : \sigma_p$ holds.

$$\frac{}{\underline{\mathsf{int}} \overset{typ}{\rightsquigarrow}_\delta \mathsf{int}} \qquad \frac{}{\underline{\mathsf{cond}} \overset{typ}{\rightsquigarrow}_\delta \mathsf{cond}^\mu} \qquad \frac{\underline{\sigma} \overset{typ}{\rightsquigarrow}_\delta \sigma}{\underline{\sigma}_p\ \mathsf{ref}^{rt} \overset{typ}{\rightsquigarrow}_\delta \sigma_p^\mu\ \mathsf{ref}^{rt}}$$

$$\frac{\mathcal{G}^- \overset{typ}{\rightsquigarrow}_\delta \Gamma^- \qquad \mathcal{G}^+ \overset{typ}{\rightsquigarrow}_\delta \Gamma^+}{\mathcal{G}^-, U \xrightarrow{p} \mathcal{G}^+ \overset{typ}{\rightsquigarrow}_\delta \Gamma^-, K^-, U \xrightarrow{p,\mu} \Gamma^+, K^+}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathcal{G}) = \mathrm{dom}(\Gamma) \\ \forall y \in \mathrm{dom}(\mathcal{G}).\ \mathcal{G}(y) \overset{typ}{\rightsquigarrow}_\delta \Gamma(y) \\ \forall l \in \mathrm{dom}(\mathcal{G}).\ \mathcal{G}(l) = (\underline{\sigma}_p, rt) \wedge p \not\leq L \implies \delta(l) \neq N\end{array}}{\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma}$$

Figure 10. Type translation rules

The judgment for translating base types is $\underline{\sigma} \overset{typ}{\rightsquigarrow}_\delta \sigma$. It states that an IMPs base type $\underline{\sigma}$ is translated to an IMPe base type σ. It is parametrized by δ to ensure that type environments for functions types are translated appropriately.

Figure 10 shows the type translation. In the rule for type environments, premise $\forall l \in \mathrm{dom}(\mathcal{G}).\ \mathcal{G}(l) = (\underline{\sigma}_p, rt) \wedge p \not\leq L \implies \delta(l) \neq N$ ensures that all confidential locations have appropriate enclave assignments (even if these locations are not used by the program).

Figure 11 shows the translation for expressions. Translation for expressions proceeds by first translating the types. They enforce the invariant that a location in enclave E_i is accessed in the same mode E_i. Rules TR-INT, TR-VAR, TR-LOC, TR-CND, and TR-OP are straightforward.

Rule TR-DEREF translates a dereference expression, The premise $\mu' \neq N \implies \mu = \mu'$ generates a conditional constraint such that if the dereferenced location is in an enclave ($\mu' \neq N$) then the expression is evaluated in the same enclave ($\mu = \mu'$). Similarly, rule TR-ISUNSET ensures that if a condition location is dereferenced, then the mode in which the expression is evaluated is appropriate.

Rule TR-FUNCTION requires that if the post type environment Γ^+ has any variables with policies more restrictive than L ($\neg\mathsf{isVarLowContext}(\Gamma^+)$), then the function is defined in an enclave ($\mu \neq N$). Intuitively, any data left in variables at the end of the function invocation may be observable by the code that invoked the function. If that data includes confidential information, then the function should not be invoked by non-enclave code.

Figure 12 shows the inference rules for translating commands. Most of these rules are straightforward and closely follow the premises of the corresponding typing rules. Premise $\mu \notin K$ occurs in many of the rules, and ensures that code in killed enclaves cannot be executed.

Rule TR-SEQ drives the entire translation. Intuitively, given a sequence $c_1; \ldots; c_n$, it translates each sub-command c_i by assigning them a different mode variable μ_i. If the translation infers that $\mu_0 = N$ but $\mu_i \neq N$, then the translated sub-command c_i' is placed inside an enclave. Variable K_i is the kill set immediately before the execution of c_i', K_i' is the kill set immediately after the execution of c_i', and K_i'' is the set of enclaves (if any) that can be safely killed after executing c_i'. Thus, we have that $K_{i+1} = K_i' \cup K_i''$.

Constraint $K_i' \cap K_i'' = \emptyset$ ensures that an enclave is not killed more than once. Constraint $\mu_0 \neq N \implies (\mu_0 = \mu_i \wedge K_i'' = \emptyset)$ states that if sequence executes entirely in an enclave ($\mu_0 \neq N$) then all sub-commands are in the same enclave and no enclaves are killed. Constraint $\mu_i \neq N \wedge \mu_i = \mu_{i+1} \implies K_i'' = \emptyset$ ensures that no enclave can be killed between sequences executing in same enclave.

Constraint $\mu_i \neq N \wedge (\mu_i \neq \mu_{i+1} \vee K_i'' \neq \emptyset) \implies \mathsf{isVarLowContext}(\Gamma_i)$ ensures that if execution exits an enclave (i.e., if command c_i' executes in an enclave, but the sequence itself is not in an enclave) then no variables contain confidential information when the enclave exits. This is required to enforce typing rule T-ENCLAVE. Notice that an enclave exit after command c_i does not necessarily mean that $\mu_{i+1} = N$. It may signal the start of a different enclave, hence we also state $\mu_i \neq \mu_{i+1}$ in the antecedent.

Rule TR-SEQ uses utility function processSeqOutput which inserts enclave and kill commands appropriately into the translation. Intuitively, enclave is introduced for a command c_i' if there is a mode change. Command kill(j) is inserted after command c_i' if $j \in K_i''$. Pseudo code for processSeqOutput is presented in Appendix C.

Rule TR-IF-ELSE requires that same sets of enclaves are killed in both the branches. Also, if variables contain confidential information on exit of either branch, then the outer mode should not be normal. Rule TR-IF-ISUNSET always places the command in an enclave to ensure that the premises of typing rule T-IF-ISUNSET are met. Rule TR-WHILE requires that no enclave is killed in the loop body. Rule TR-CALL requires that if set U is non-empty, then the function is defined in an enclave.

Soundness of Translation Successful translation of well-typed IMPs program produces a well-typed IMPe program.

Theorem 2 (Soundness of Translation)**.** *Let $\mathcal{G}$ be a well-typed IMPs environment and Γ be an IMPe environment that is well-typed for δ. For all commands $c \in$ IMPs, if $pc, \mathcal{G}, U \vdash c\colon \mathcal{G}'$ and $pc, \mathcal{G}, K, c, \mathcal{G}' \rightsquigarrow \mu, \Gamma, U, \delta, c', \Gamma', K'$ then $pc, \mu, \Gamma, K, U \vdash_\delta c' : \Gamma', K'$*

Given a well-typed IMPs program and environment $\mathcal{G}$, if the translation is successful using an IMPe environment Γ that is well-typed for δ, then the translated IMPe program is also well-typed. Since a well-typed IMPe program is secure (Theorem 1), the translation thus guarantees security against all the attackers described earlier.

Note that well-typedness of the translated program is contingent on the success of translation. The only way that translation will fail is if predicate $\mathsf{isVarLowContext}(\Gamma')$ does not

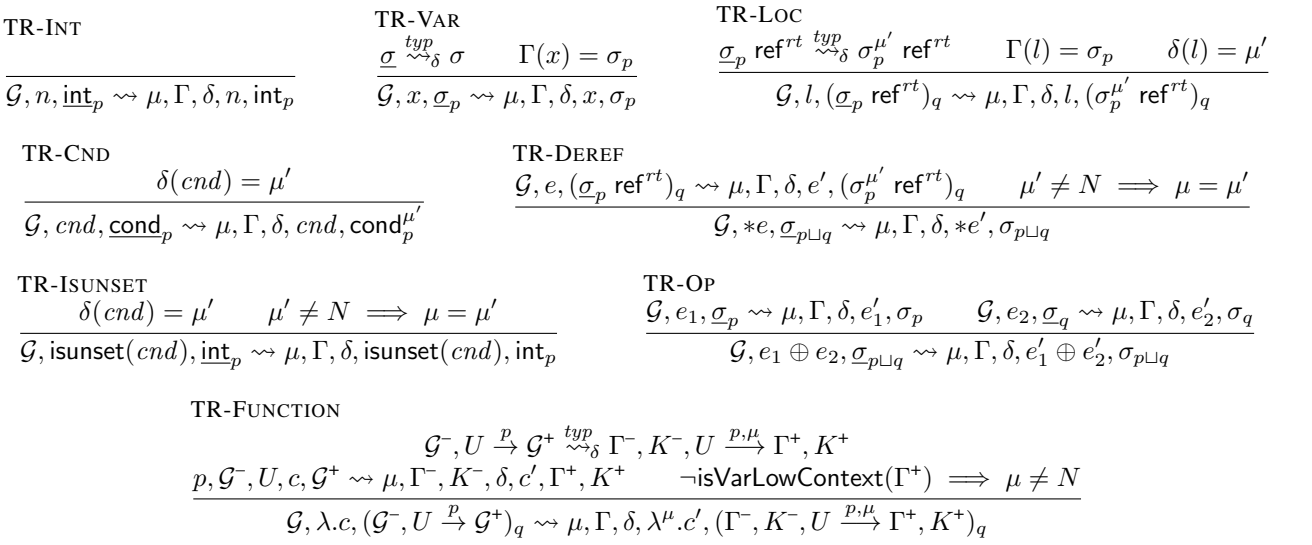

Figure 11. Translation of expressions

hold for Γ' at enclave exit, i.e., a variable contains confidential information when the enclave exits. This is because the trivial solution of putting all code and data inside a single enclave will succeed provided there is no confidential information left in variables at the end of the program. One approach to ensure translation always succeeds is to add a semantics-preserving transformation that zeroes-out variables as soon as they are dead.

6.2 Constraint Solution and Optimization

The constraints used in the translation of IMPs programs to IMPE can be expressed as a Boolean SAT instance, assuming that the mode set $Mode = \{N, E_1, E_2, \dots\}$ is of a fixed finite size. Specifically, the constraints restrict modes of locations and code, and kill sets (which are sets of enclaves). All constraints generated during translation can be encoded straightforwardly in a SAT formula. For a program of size n with m locations where $|Mode| = k$, the size of SAT formula is $O((n+m)^2 + nk)$.[3]

There may be many possible translations of a given IMPs program without any of them being clearly the "best" translation. Naively, we could try to place the entire program and all locations in a single enclave. However, even if successful, this is not always desirable for at least two reasons. First, an enclave may have size restrictions and a program can be too large to fit.[4] Second, even if the program can fit inside an enclave, it may be desirable to have as little code as possible in enclaves, to reduce the trusted computing base (i.e., the code that must be assumed to be correct; security for a non-enclave active attacker assumes that enclave code does not contain exploitable vulnerabilities).

There are several possible criteria for comparing the quality of translations, including minimizing the code and data inside enclaves (which corresponds to minimizing the trusted computing base (TCB)), reducing the lifetime of confidential data by killing enclaves as soon as possible, or minimizing the performance penalty of enclaves.[5]

We can cast our translation as a constraint optimization problem that optimizes an objective function that approximates TCB-size, lifetime of enclaves, performance penalty, or a combination of these.

A *pseudo-Boolean function* $f : \{0,1\}^n \rightarrow \mathbb{R}$ is a real-valued function of a finite number of 0-1 valued variables [8]. A *pseudo-Boolean constraint* is an equality or inequality between pseudo-Boolean functions. *Pseudo-Boolean optimization* (PBO) optimizes a pseudo-Boolean function subject to pseudo-Boolean constraints. PBO is 0-1 multilinear integer programming and is NP-hard [8]. We can encode the SAT formula for a translation as a pseudo-Boolean constraint and express TCB size and performance as pseudo-Boolean functions to be minimized.

We can compute the TCB cost by counting the number of non-sequence commands placed in enclaves.

Killing an enclave as soon as possible reduces the window of vulnerability. This can be achieved by maximizing the size of kill sets at all program points, which effectively kills enclaves as soon as possible. Moreover, we can facilitate killing enclaves as early as possible by using more en-

[3] Intuitively, $(n+m)$ mode variables, pairs of which are constrained to be either equal or different are generated, resulting in at most $(n+m)^2$ constraints. Additionally, nk kill set constraints (e.g., $K_{i+1} = K_i' \cup K_i''$ in TR-SEQ) are generated. Thus the size of the SAT formula is $O((n+m)^2 + nk)$.

[4] On some models, SGX enclaves have a maximum size of 2^{31} bits [25].

[5] In Intel SGX, entering or exiting an enclave flushes all TLB entries [25].

TR-SKIP

$$\frac{\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mu \notin K}{pc, \mathcal{G}, U, \mathsf{skip}, \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{skip}, \Gamma, K}$$

TR-ASSIGN

$$\frac{\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}, e, \underline{\sigma}_q \rightsquigarrow \mu, \Gamma, \delta, e', \sigma_q \qquad (pc \sqcup q) \not\leq L \implies \mu \neq N \qquad \mu \notin K}{pc, \mathcal{G}, U, x := e, \mathcal{G}[x \mapsto \underline{\sigma}_{pc \sqcup q}] \rightsquigarrow \mu, \Gamma, K, \delta, x := e', \Gamma[x \mapsto \sigma_{pc \sqcup q}], K}$$

TR-DECLASSIFY

$$\frac{\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}, e, \underline{\sigma}_q \rightsquigarrow \mu, \Gamma, \delta, e', \sigma_q \qquad (pc \sqcup q) \not\leq L \implies \mu \neq N \qquad \mu \notin K}{L, \mathcal{G}, U, x := \mathsf{declassify}(e), \mathcal{G}[x \mapsto \underline{\sigma}_L] \rightsquigarrow \mu, \Gamma, K, \delta, x := \mathsf{declassify}(e'), \Gamma[x \mapsto \sigma_L], K}$$

TR-OUTPUT

$$\frac{\mathcal{G}, e, \underline{\sigma}_p \rightsquigarrow \mu, \Gamma, \delta, e', \sigma_p \qquad \mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mu \notin K}{pc, \mathcal{G}, U, \mathsf{output}\ e\ \mathsf{to}\ \ell, \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{output}\ e'\ \mathsf{to}\ \ell, \Gamma, K}$$

TR-SETCND

$$\frac{\delta(cnd) = \mu' \qquad \mu' \neq N \implies \mu = \mu' \qquad pc = L \qquad \mu \notin K}{pc, \mathcal{G}, U, \mathsf{set}(cnd), \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{set}(cnd), \Gamma, K}$$

TR-SEQ

$$\frac{\begin{array}{c}\mathcal{G}_i \overset{typ}{\rightsquigarrow}_\delta \Gamma_i \qquad pc, \mathcal{G}_{i-1}, U, c_i, \mathcal{G}_i \rightsquigarrow \mu_i, \Gamma_{i-1}, K_i, \delta, c_i', \Gamma_i, K_i' \\ \forall i \in \{1 \ldots n\}.\ K_{i+1} = K_i' \cup K_i'' \qquad K_i'' \cap K_i' = \emptyset \\ \mu_0 \neq N \implies (\mu_0 = \mu_i \wedge K_i'' = \emptyset) \qquad \mu \notin K_1 \\ \mu_i \neq N \wedge \mu_i = \mu_{i+1} \implies K_i'' = \emptyset \\ \mu_i \neq N \wedge (\mu_i \neq \mu_{i+1} \vee K_i'' \neq \emptyset) \implies \mathsf{isVarLowContext}(\Gamma_i) \\ c' = \mathsf{processSeqOutput}(\vec{c}_{1:n}', \mu_0, \vec{\mu}_{1:n}, \vec{K''}_{1:n})\end{array}}{pc, \mathcal{G}_0, U, c_1; \ldots; c_n, \mathcal{G}_n \rightsquigarrow \mu_0, \Gamma_0, K_1, \delta, c', \Gamma_n, K_{n+1}}$$

TR-UPDATE

$$\frac{\begin{array}{c}\mathcal{G}, e_1, \underline{\sigma}_p\ \mathsf{ref}_q^{rt} \rightsquigarrow \mu, \Gamma, \delta, e_1', \sigma_p^{\mu'}\ \mathsf{ref}_q^{rt} \\ \mathcal{G}, e_2, \underline{\sigma}_{p'} \rightsquigarrow \mu, \Gamma, \delta, e_2', \sigma_{p'} \\ \mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mu' \neq N \implies \mu = \mu' \qquad \mu \notin K\end{array}}{pc, \mathcal{G}, U, e_1 \leftarrow e_2, \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, e_1' \leftarrow e_2', \Gamma, K}$$

TR-IF-ELSE

$$\frac{\begin{array}{c}\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}, e, \underline{\mathsf{int}}_p \rightsquigarrow \mu, \Gamma, \delta, e', \mathsf{int}_p \qquad pc', \mathcal{G}, U, c_1, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, c_1', \Gamma', K' \\ pc', \mathcal{G}, U, c_2, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, c_2', \Gamma', K' \qquad pc \sqcup p \leq pc' \qquad (\neg \mathsf{isVarLowContext}(\Gamma') \vee p \not\leq L) \implies \mu \neq N \qquad \mu \notin K\end{array}}{pc, \mathcal{G}, U, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{if}\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \Gamma', K'}$$

TR-IF-ISUNSET

$$\frac{\begin{array}{c}\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}, \mathsf{isunset}(cnd), \underline{\mathsf{int}}_L \rightsquigarrow \mu, \Gamma, \delta, \mathsf{isunset}(cnd), \mathsf{int}_L \\ pc, \mathcal{G}, U \cup \{cnd\}, c_1, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, c_1', \Gamma', K' \qquad pc, \mathcal{G}, U, c_2, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, c_2', \Gamma', K' \qquad \mu \notin \{N\} \cup K\end{array}}{pc, \mathcal{G}, U, \mathsf{if}\ \mathsf{isunset}(cnd)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mathcal{G}' \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{if}\ \mathsf{isunset}(cnd)\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \Gamma', K'}$$

TR-WHILE

$$\frac{\begin{array}{c}\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}, e, \underline{\mathsf{int}}_p \rightsquigarrow \mu, \Gamma, \delta, e', \mathsf{int}_p \qquad pc', \mathcal{G}, U, c, \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, c', \Gamma, K \\ (\neg \mathsf{isVarLowContext}(\Gamma) \vee p \not\leq L) \implies \mu \neq N \qquad pc \leq pc' \qquad \mu \notin K\end{array}}{pc, \mathcal{G}, U, \mathsf{while}\ e\ \mathsf{do}\ c, \mathcal{G} \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{while}\ e'\ \mathsf{do}\ c', \Gamma, K}$$

TR-CALL

$$\frac{\begin{array}{c}\mathcal{G} \overset{typ}{\rightsquigarrow}_\delta \Gamma \qquad \mathcal{G}_{out} \overset{typ}{\rightsquigarrow}_\delta \Gamma_{out} \\ \mathcal{G}, e, \mathcal{G}^-, U \overset{p}{\rightarrow} \mathcal{G}^+ \rightsquigarrow \mu, \Gamma, \delta, e', \Gamma^-, K^-, U \overset{p,\mu}{\longrightarrow} \Gamma^+, K^+ \\ \forall y \in \mathrm{dom}(\Gamma^+), \Gamma^+(y) \leq \Gamma_{out}(y) \\ \forall y \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^+).\ \Gamma(y) = \Gamma_{out}(y) \\ \forall y \in \mathrm{dom}(\Gamma^-), \Gamma(y) \leq \Gamma^-(y) \qquad U \neq \emptyset \implies \mu \neq N \\ K = K^- \qquad K_{out} = K^+ \qquad \mu \notin K\end{array}}{pc, \mathcal{G}, U, \mathsf{call}(e), \mathcal{G}_{out} \rightsquigarrow \mu, \Gamma, K, \delta, \mathsf{call}(e'), \Gamma_{out}, K_{out}}$$

TR-SUB

$$\frac{\begin{array}{c}\mathcal{G}_1 \overset{typ}{\rightsquigarrow}_\delta \Gamma_1 \qquad \mathcal{G}_1' \overset{typ}{\rightsquigarrow}_\delta \Gamma_1' \\ \mathcal{G}_2 \overset{typ}{\rightsquigarrow}_\delta \Gamma_2 \qquad \mathcal{G}_2' \overset{typ}{\rightsquigarrow}_\delta \Gamma_2' \\ \Gamma_2 \leq \Gamma_1 \qquad \Gamma_1' \leq \Gamma_2' \qquad pc_2 \leq pc_1 \\ pc_1, \mathcal{G}_1, U, c, \mathcal{G}_1' \rightsquigarrow \mu, \Gamma_1, K, \delta, c', \Gamma_1', K\end{array}}{pc_2, \mathcal{G}_2, U, c, \mathcal{G}_2' \rightsquigarrow \mu, \Gamma_2, K, \delta, c', \Gamma_2', K}$$

Figure 12. Translation for commands

claves, i.e., partitioning code and data into enclaves at fine granularity. This is also optimized by maximizing the size of kill sets. Note that to avoid spuriously putting public data into enclaves to increase the total number of enclaves that can be killed, we require that each killed enclave has at least some confidential data stored in it.

Enclave entry and exit is expensive and penalizes the run-time performance. Although we have not implemented it, we could approximate the run-time cost using a Control Flow Graph (CFG) and approximating how frequently execution enters and exits enclaves.

7. Comparison with SGX

Although there are several hardware-enforced enclave-like mechanisms, IMPE is most heavily influenced by SGX. We discuss how IMPE relates to SGX.

First, we assume that enclaves are isolated from each other: code in enclave E_i can not access memory in enclave E_j when $i \neq j$. SGX does enforce this via an access control mechanism, but uses a single encryption key to protect the contents of *all* enclaves. Some enclave mechanisms (such as TrustZone) do not provide multiple enclaves.

Second, we assume that once an enclave is killed, the contents of the enclave can never be recovered, thus providing forward secrecy. However, the current design of SGX bases access control decisions on the *initial measurement* of an enclave. That is, if another enclave is created that has the exact same initial contents as the killed enclave, a replay attack may be possible, whereby the new enclave decrypts memory pages from the killed enclave.

Third, our model assumes inputs to an execution are provided in the initial memory and output channels exist for security levels L and H. Our model can be easily modified to use channels for input instead of the initial memory. Secure channels from an SGX enclave to remote parties can be straightforwardly implemented using cryptographic mechanisms. However, SGX currently provides little support for secure output to local devices and no support for secure local input, possibly making it unsuitable for, e.g., securely checking a locally-entered password. However, support for secure local I/O is emerging, such as TrustZone's Trusted User Interface [17]. This supports our modeling choice to allow the enclave to receive and send confidential information, which can represent (remote or local) secure I/O.

8. Evaluation

We implement six case studies (many inspired by related work [7, 16, 34, 36]) to evaluate the expressiveness of security policies, and the translation from IMPS to IMPE. The translator and case studies are available online [19]. All case studies are implemented as (well-typed) IMPS programs which translate successfully to IMPE programs. Thus all case studies are secure against passive, enclave, and non-enclave active attackers.

We extend the calculi with strings, pairs, and arrays. The types of IMPE and IMPS are extended as follows.

$$\sigma ::= \cdots \mid \mathsf{string} \mid \sigma_1 \times \sigma_2 \mid \tau^{\mu}\ []^{rt}$$
$$\underline{\sigma} ::= \cdots \mid \underline{\mathsf{string}} \mid \underline{\sigma}_1 \times \underline{\sigma}_2 \mid \underline{\tau}\ []^{rt}$$

An array is a sequence of locations with the constraint that all elements of the array are in the same enclave (or all elements are in no enclave). IMPE array type $\tau^{\mu}\ []^{rt}$ indicates an array that contains values of type τ, mode μ indicates in which enclave (if any) the array is placed and rt indicates if the contents of array are mutable. The IMPS array type is similar except that there is no mode annotation. Types for strings and pairs are straightforward.

Password Authentication Recall the password authentication example (with declassification) from Section 3.2. Consider an IMPS version (i.e., without any enclave annotations). Translating it with our tool gives the following.

```
enclave(1, status := declassify(*password = *guess));
kill(1);set(end); output status to L
```

The translation assigns enclave E_1 to locations password and guess (i.e., translated locations have types password : $\mathsf{int}_H^{E_1}\ \mathsf{ref}^{\mathsf{immut}}$ and guess : $\mathsf{int}_L^{E_1\ \mathsf{end}\nearrow\top}\ \mathsf{ref}^{\mathsf{immut}}$). The declassification is placed inside E_1 because it reads password and guess. The translation kills the enclave immediately after exiting the enclave. This is as early as possible, thus minimizing the window of vulnerability.

Private Browsing A private session of a web browser allows a user to browse the web with the assurance that the browsing history cannot be retrieved after the private session has ended. However, private browsing implementations are error prone, and many leak information from private sessions [1, 33]. We model a private browsing session where the user starts a private session, browses, then ends the session. The security requirement can be expressed as an erasure policy that states that all private browsing data (and data derived from it) should be erased when a condition marking the end of the session is set.

Since our calculi model input as the initial memory, we assume that the initial memory contains the user's input to the private session (e.g., an array of URLs to visit). The user's input has erasure policy $H\ ^{\mathsf{end}}\!\!\nearrow\top$, where condition end is set at the end of the private session. During the private session, output is sent to channel H. Once the session ends, we model normal browsing by output to channel L.

Translation assigns enclave E_1 to all the locations containing the user's input to the private session. It also places all code related to the private browsing session inside enclave E_1 and generates a kill instruction before resuming normal browsing.

Secure Calculator We implement a secure calculator that performs public operations on confidential data. This is

a model of, for example, a tax computation, where well-known operations (the tax computation) are performed on confidential input (an individual's financial information). The operations are chosen dynamically (i.e., public inputs specify which operations to perform). The result of the computation is output to channel H. The initial memory contains an array of operations to perform (with security policy L), and a stack of confidential data (with security policy H). The program iterates over the array of operations, performing them on the stack of data. The following snippet is illustrative of the code for this case study.

```
while (i < numOps)
    if (*ops[i] = "add") then
        stack[top − 1] ← *stack[top − 1] + *stack[top];
    else if (*ops[i] = "sub") then
        ...
    top := top − 1;
    output *stack[top] to H;
    i := i + 1
```

The translation places the confidential data in enclave E_1 and the array of operations outside the enclave. To minimize the TCB, the translation places in the enclave each command that reads or writes the confidential stack, but leaves all other commands outside the enclave. For example:

```
if (*ops[i] = "add") then
    enclave(1, stack[top − 1] ←
                        *stack[top − 1] + *stack[top]);
```

This reduces the amount of code in enclaves, but will likely result in poor performance, due to frequent enclave entries and exits. If this is a concern, the translation could use a different objective function that balances estimated performance with TCB size.

Secure Map-Reduce We model a word-count program that takes a set of private documents, and computes word frequencies, similar to a case study by Schuster et al. [34]. The program follows the map-reduce model, in which partial counts of each word in a document are first emitted and the partial counts are then combined. Each document is modeled as an array of confidential strings $\mathsf{doc} : \mathsf{string}_H\ []^{\mathsf{immut}}$, and the initial memory contains several such documents. A map function takes a document as input and produces counts of each word in the document; a reduce function takes as input a specific word, and sums the partial counts of the word. The output of the program $\mathsf{wordcount} : (\mathsf{string} \times \mathsf{int})_H[]^{\mathsf{mut}}$ is an array of pairs of words and the frequency of that word.

The translation places the entire map and reduce functions inside an enclave, as well as all of the documents. That is, the entire computation is placed inside an enclave, as is the hand-coded map-reduce computation of Schuster et al. The enclave is killed after map-reduce computation.

Secure Query Processing We model query processing over confidential data, similar to the *Query Processing over Encrypted Database* case study by Sinha et al. [36]. Given a database table with public keys (e.g., employee names, in the column name) and confidential data (e.g., wage payments, in column wages), the query selects rows that match a given key, and sums the confidential data. The selection of rows uses only non-confidential data but the subsequent summing uses confidential data.

We model columns name and wages as arrays, with policies L and H respectively. Row selection chooses all indices of array name that are equal to key "alice". Summing computes the sum of all wages corresponding to the selected indices.

The translation places array wages in enclave E_1, but leaves array name outside of any enclave. The row selection computation is placed outside an enclave, and the summing operation is placed inside enclave E_1. Our automated translation places the same data and computation in enclaves as the (manually coded) case study of Sinha et al.

Secure Chat Client We model a secure chat client, inspired by the case study of Askarov et al. [7]. Messages sent and received by the client are emitted to a log. When the user enters a "clear" command, all messages (including the log) should be erased. We model messages sent and received and commands entered by the user as data in the initial memory. We model logging as an update to location log. We give messages and the log the erasure policy $L \overset{\mathsf{clear}}{\nearrow} \top$, which states that the contents of log are erased when condition clear is set. The condition is set only when a "clear" command is issued. The translation places log in an enclave, as well as all code that updates the log.

9. Related Work

Models for Secure Hardware Architecture and Compilation Fournet and Planul [16] securely compile imperative programs into distributed programs using cryptography and hardware mechanisms (such as Trusted Platform Modules (TPM) and secure boot) to enforce noninterference for confidentiality and integrity. They emulate secure memory (that cannot be accessed by adversaries) and enforce control-flow restrictions on the distributed program. The compiled program is proven to be at least as secure as the source program: for every attack on the compiled program there is a corresponding attack on the source program, with the same information leakage. By contrast, we focus on expressive security policies (erasure and declassification) that go beyond noninterference. Their system doesn't provide erasure guarantees. We target a single machine and use enclave mechanisms that directly provide secure memory (instead of emulating secure memory via cryptographic mechanisms). Both our work and theirs shield the programmer from the mechanisms used to enforce security. Although we do not focus on integrity guarantees in this work, we rely on enclaves to provide integrity guarantees on code running inside enclaves (cf. security against non-enclave active attackers). We

believe that our target calculus IMPE can be extended to provide integrity guarantees about computation inside enclaves.

VC3 [34] enables distributed map-reduce computations in untrusted cloud environments while keeping code and data secret, using SGX enclaves to protect against adversaries that might control the entire software stack of the cloud provider's infrastructure. We instead focus on providing confidentiality for general programs in the presence of an attacker controlling the entire software stack of a single system. In VC3, all data is confidential (i.e., equivalent to our policy H) and all map-reduce computation of a given node is placed inside a single enclave on that node. By contrast, we use expressive security policies (i.e., declassification and erasure) and infer enclave placement to optimize various objective functions. VC3 ensures that only address-taken variables are read and written. The *region-self-integrity* mechanism prevents unintended disclosure of information due to low-level errors (e.g., buffer overflow). This can be used as a defense-in-depth mechanism in our work to reduce the possibility of an enclave active attacker exploiting vulnerabilities in enclave code.

Moat [36] models SGX using BoogiePL [13] and verifies the confidentiality of binary SGX programs in the presence of "havocing" adversaries capable of modifying non-enclave code. A havocing adversary is analogous to our non-enclave attacker, which can arbitrarily modify non-enclave code. Thus, ensuring confidentiality against a havocing adversary corresponds to security for a non-enclave active attacker. Our work also considers enclave active attackers, which are more powerful than havocing adversaries. Our work differs from Moat in shielding developers from low-level enclave-specific details.

Ironclad [23] provides verifiable remote equivalence: an application running on an untrusted server is indistinguishable from its high-level abstract state machine. Ironclad uses secure hardware (e.g., TPM) as the root of trust and to enable secure channels from verified software to remote clients. Our work could potentially be used in an Ironclad-like setting to reduce verification effort: enclave inference can be used to identify and minimize the security-critical parts of an application, which reduces the code that must be verified.

Sinha et al. [37] enforce confidentiality by placing an entire application inside an SGX enclave and restricting its communication with external memory through a narrow interface to a trusted library. They enforce *Information Release Confinement*, which ensures that the application satisfies a form of control-flow integrity and never directly accesses non-enclave memory. This work is complementary to ours, and could be used for defense-in-depth in our work, making it harder for an enclave active attacker to exploit vulnerabilities in enclave code.

Patrignani et al. [27] provide a fully-abstract secure compilation scheme for compiling strongly typed object-oriented languages to protected module architectures (PMAs) that offer memory isolation mechanisms and are similar to enclaves. Objects containing private methods are placed inside protected modules thus preventing a low-level attacker from bypassing encapsulation mechanisms. The compilation scheme is proven to preserve and reflect the encapsulation of the source program. Their low-level attacker is similar to the non-enclave active attacker in our model. Though we do not aim for full abstraction, our work provides a stronger information flow guarantee for applications with more expressive security requirements against different attackers.

None of the above works consider applications using multiple enclaves whereas our programming model supports multiple enclaves seamlessly.

Language-based Information-Flow Control Much work in language-based information-flow control is concerned with enforcing application-specific security guarantees [14, 26, 29, 38] Our work extends these techniques to a setting where the underlying software stack is not trusted. That is, we consider strong low-level attackers that are capable of arbitrary corruption of some parts of a program.

Information Erasure A key emphasis in our work is the enforcement of information erasure using SGX-like mechanisms. Information erasure is related to data deletion, but requires that the observable behavior of a system reveals nothing about the deleted data, which may, for example, require tracking and deletion of data derived from the deleted data. Language-based information erasure was introduced by Chong and Myers [9], and several works present techniques for enforcing erasure (e.g., [7, 10, 24]). By contrast with these previous language-based approaches, we protect against more powerful lower-level attackers.

Other work uses language- and system-based techniques to ensure data deletion at the system- or architectural-level of abstraction. Chow et al. [11] enforce data deletion by analyzing the lifetime of sensitive data, and automatically zeroing out data in memory. Perlman [28] proposes a file system that uses cryptographic techniques to reliably delete files. These approaches may fail to remove derived data, and thus will not enforce information erasure. Lacuna [15] runs sensitive computations in a "private session" and can securely delete all session data at the end of the session (including data used to communicate with peripheral devices). Provided all sensitive information is contained within a private session, Lacuna can enforce both data deletion and information erasure.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1054172.

References

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[2] Apple. iOS security. https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Sept. 2015.

[3] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 191–205, 2012.

[4] ARM. ARM security technology — building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[5] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 43–59, 2009.

[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, Oct. 2008.

[7] A. Askarov, S. Moore, C. Dimoulas, and S. Chong. Cryptographic enforcement of language-based erasure. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, July 2015.

[8] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, Nov. 2002.

[9] S. Chong and A. C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, pages 241–254, 2005.

[10] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111, June 2008.

[11] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, 2005.

[12] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[13] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.

[14] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[15] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 61–75, 2012.

[16] C. Fournet and J. Planul. Compiling information-flow security to minimal trusted computing bases. In *Proceedings of the 20th European Conference on Programming Languages and Systems*, pages 216–235, 2011.

[17] GlobalPlatform. Trusted user interface API specification v1.0. http://www.globalplatform.org/specificationsdevice.asp, 2013.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

[19] A. Gollamudi. Impslator. https://github.com/anithag/impslator, June 2016.

[20] A. Gollamudi and S. Chong. Automatic enforcement of expressive security policies using enclaves. Technical Report TR-2-2016, Harvard University, Aug. 2016.

[21] P. Gutmann. Data remanence in semiconductor devices. In *The Tenth USENIX Security Symposium Proceedings*, pages 39–54, 2001.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[23] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2014.

[24] S. Hunt and D. Sands. Just forget it—the semantics and enforcement of information erasure. In *Proceedings of the 17th European Symposium on Programming*, pages 239–253, 2008.

[25] Intel. Intel software guard extensions (Intel SGX) programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[26] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, June 2004.

[27] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6, 2015.

[28] R. Perlman. File System Design with Assured Delete. In *Proceedings of the Third IEEE International Security in Storage Workshop*, pages 83–88, 2005.

[29] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[30] A. Sabelfeld and A. C. Myers. A model for delimited release. In *Proceedings of the 2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191, 2004.

[31] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[32] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM Trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80, 2014.

[33] K. Satvat, M. Forshaw, F. Hao, and E. Toreini. On the privacy of private browsing - a forensic approach. *Journal of Infor-*

mation Security and Applications, 19(1), Feb. 2014.

[34] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 38–54, 2015.

[35] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 45–48, 2016.

[36] 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, 2015.

[37] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 665–681, 2016.

[38] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3): 167–187, Jan. 1996.

A. Enclave equivalence $\overset{enc}{\simeq}$

Equivalence relation $\overset{enc}{\simeq}$ is used to characterize the ability of an active attacker. Intuitively, given commands c and c', we have $c \overset{enc}{\simeq} c'$ if c and c' have the same code in enclaves, but may differ arbitrarily on non-enclave code. We define $\overset{enc}{\simeq}$ using function χ which syntactically extracts enclave code.

Definition 6 (Enclave Equivalence)**.** *Two IMP*E *programs* c_1 *and* c_2 *are enclave equivalent, denoted* $c_1 \overset{enc}{\simeq} c_2$*, iff*

$$\chi(c_1) = \chi(c_2)$$

where

$$\begin{aligned}\chi(\mathsf{enclave}(j, c)) &= \{(E_j, c)\}\\ \chi(\lambda^{E_j}.c) &= \{(E_j, \lambda^{E_j}.c)\}\\ \chi(\lambda^{N}.c) &= \chi(c)\end{aligned}$$

and all atomic expressions and commands return the empty set, e.g.,

$$\begin{aligned}\chi(\mathsf{skip}) &= \emptyset\\ \chi(n) &= \emptyset\end{aligned}$$

and all other expressions and commands recurse on sub-expressions and sub-commands, e.g.,

$$\begin{aligned}\chi(c_1; \ldots; c_n) &= \chi(c_1) \cup \cdots \cup \chi(c_n)\\ \chi(e_1 \oplus e_2) &= \chi(e_1) \cup \chi(e_2)\end{aligned}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\ell_1 \leq \ell_2} \qquad \frac{p_1 \leq \ell_2}{p_1 \leq \ell_2 \overset{cnd}{\nearrow} \ell_2'} \qquad \frac{\ell_1' \leq p_2}{\ell_1 \overset{cnd}{\nearrow} \ell_1' \leq p_2}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \qquad \ell_1' \sqsubseteq \ell_2'}{\ell_1 \overset{cnd}{\nearrow} \ell_1' \leq \ell_2 \overset{cnd}{\nearrow} \ell_2'} \qquad \frac{\sigma \leq \sigma' \qquad p \leq q}{\sigma_p \leq \sigma_q'} \qquad \frac{}{\sigma \leq \sigma}$$

$$\frac{\Gamma_2^- \leq \Gamma_1^- \qquad p_2 \leq p_1 \qquad \Gamma_1^+ \leq \Gamma_2^+}{\Gamma_1^-, K^-, U \xrightarrow{p_1, \mu} \Gamma_1^+, K^+ \leq \Gamma_2^-, K^-, U \xrightarrow{p_2, \mu} \Gamma_2^+, K^+}$$

Figure 13. Policy ordering and subtyping

For example, given $c_1 = \mathsf{enclave}(1, \mathsf{output}\ 42\ \mathsf{to}\ L)$ and $c_2 = l \leftarrow 1; \mathsf{enclave}(1, \mathsf{output}\ 42\ \mathsf{to}\ L)$, we have $c_1 \overset{enc}{\simeq} c_2$, since $\chi(c_1) = \chi(c_2) = \{(E_1, \mathsf{output}\ 42\ \mathsf{to}\ L)\}$.

B. IMPE Type System

Figure 13 defines the relabeling relation $\leq$ on policies.

C. Pseudo Code

Function processKill inserts $\mathsf{kill}(j)$ whenever an enclave E_j is killed and function processSeqOutput wraps the largest sequence of code with mode E_j in $\mathsf{enclave}(j, \ldots)$

$\mathsf{processKill}(K) =$
  match K with
    $|k \cup K' \rightarrow \mathsf{kill}(k); \mathsf{processKill}(K')$
    $|\emptyset \rightarrow ()$
$\mathsf{processSeqOutput}(\vec{c}'_{1:z}, \mu_0, \vec{\mu}_{1:z}, \vec{K}_{1:z}) =$
  match $(\mu_0, \vec{\mu}_{1:z}, \vec{K}_{1:z})$ with
    $| \mu', (\mu' \ldots \mu'), (\emptyset \ldots \emptyset) \rightarrow c'_1; \ldots; c'_z$
    $| N, (N, \vec{\mu}_{2:z}), (K', \vec{K}_{2:z}) \rightarrow$
      $c'_1; \mathsf{processKill}(K');$
      $\mathsf{processSeqOutput}(\vec{c}'_{2:z}, \mu_0, \vec{\mu}_{2:z}, \vec{K}_{2:z})$
    $| N, (E_j, \ldots, E_j, N, \vec{\mu}_{m+1:z}),$
     $(\emptyset, \ldots \emptyset, K_{m-1}, \vec{K}_{m:z}) \rightarrow$
      $\mathsf{enclave}(j, c'_1; \ldots; c'_{m-1}); \mathsf{processKill}(K_{m-1});$
      $\mathsf{processSeqOutput}(\vec{c}'_{m:z}, \mu_0, \vec{\mu}_{m:z}, \vec{K}_{m:z})$
    $| N, (E_j, \ldots, E_j, E_u, \vec{\mu}_{m+1:z}),$
     $(\emptyset, \ldots \emptyset, K_{m-1}, \vec{K}_{m:z}) \rightarrow$
      $\mathsf{enclave}(j, c'_1; \ldots; c'_{m-1}); \mathsf{processKill}(K_{m-1});$
      $\mathsf{processSeqOutput}(\vec{c}'_{m:z}, \mu_0, \vec{\mu}_{m:z}, \vec{K}_{m:z})$