Language Support for Secure Software Development with Enclaves

Aditya Oak
TU Darmstadt

Amir M. Ahmadian
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Guido Salvaneschi
University of St.Gallen

***Abstract*—Confidential computing is a promising technology for securing code and data-in-use on untrusted host machines, e.g., the cloud. Many hardware vendors offer different implementations of Trusted Execution Environments (TEEs). A TEE is a hardware protected execution environment that allows performing confidential computations over sensitive data on untrusted hosts. Despite the appeal of achieving strong security guarantees against low-level attackers, two challenges hinder the adoption of TEEs. First, developing software in high-level managed languages, e.g., Java or Scala, taking advantage of existing TEEs is complex and error-prone. Second, partitioning an application into components that run inside and outside a TEE may break application-level security policies, resulting in an insecure application when facing a realistic attacker.**

In this work, we study both these challenges. We present J_E, a programming model that seamlessly integrates a TEE, abstracting away low-level programming details such as initialization and loading of data into the TEE. J_E only requires developers to add annotations to their programs to enable the execution within the TEE. Drawing on information flow control, we develop a security type system that checks confidentiality and integrity policies against realistic attackers with full control over the code running outside the TEE. We formalize the security type system for the J_E core and prove it sound for a semantic characterization of security. We implement J_E and the security type system, enable Java programs to run on Intel SGX with strong security guarantees. We evaluate our approach on use cases from the literature, including a battleship game, a secure event processing system, and a popular processing framework for big data, showing that we correctly handle complex cases of partitioning, information flow, declassification, and trust.

***Index Terms*—Information Flow Control, Trusted Execution Environment, Robust Declassification, Security Type System**

I. Introduction

Confidential computing includes recent technologies to protect data-in-use through isolating computations to a hardware-based Trusted Execution Environment (TEE). TEEs provide hardware-supported enclaves to protect data and code from the system software. Over the past few years, an array of TEE designs has been developed, including Intel's Software Guard Extensions (SGX) [1], [2], ARM TrustZone [3], MultiZone [4] and others [5], [6], [7], [8], [9]. Using TEEs, data can be loaded securely in plain text and processed at native speed within an enclave even on a third-party machine. SGX is a TEE implementation from Intel which has been successfully used in a number of industry products [10], [11].

The issue with confidential computing Supporting confidential computing in a way that is both accessible for developers and technically secure is still an open problem.

First, **seamless integration** of enclave programming into software applications remains challenging. For example, Intel provides a C/C++ interface to the SGX enclave but no direct support is available for managed languages. As managed languages like Java and Scala are extensively used for developing distributed applications, developers need to either interface their programs with the C++ code executing in the enclave (e.g., using the Java Native Interface [12]) or compile their programs to native code (e.g., using Java Native [13]) relinquishing many advantages of managed environments.

A second aspect concerns security with **realistic attackers**. Standard security analysis of code protects against passive attackers, as common for untrusted/buggy code executing in a single trusted host [14]. Yet, with enclaves, programs run in a trusted environment within an untrusted host: the attacker can control the untrusted environment to cause additional leaks of sensitive information through the interface between the two environments. An active attacker may force the enclave program to violate the security policy by compromising the integrity of inputs at the interface or by controlling the execution order of interface components, e.g., to trigger execution paths and side effects that were not possible in the original program. Current research adopts Information Flow Control (IFC) to ensure that the code within an enclave does not leak sensitive information to the non-enclave environment [15], [16], [17]. This research, however, either takes a limited view of a passive attacker that only observes the data leaving the enclave, or it incorporates the effects of an active attacker into the execution semantics and the security condition, thus requiring additional verification effort to secure enclave programs.

These challenges lead us to the following key research questions addressed by the paper: (a) How to enable seamless integration of enclaves and managed languages like Java? (b) What is the right security model for realistic enclave attacks and how to statically verify the security of enclave programs with respect to these attacks? (c) How to harden state-of-the-art IFC tools to verify security in the TEE context? (d) How to demonstrate feasibility via realistic use cases?

Accessible and secure confidential computing To address the questions above, we present J_E, a programming model that offers language-level integration of enclave technology. We leverage IFC to secure applications running within TEEs and propose a security condition and an enforcement mechanism

targeting realistic attackers in the context of TEEs.

To support **seamless integration** of enclave programming, J_E defines a programming model for developers of enclave applications as a Java extension based on three annotations and two operators. Programmers can execute computations securely just by adding the `@Enclave` annotation to classes to be placed inside the enclave and the `@Gateway` annotation to the methods accessible from the non-enclave environment. Also, programmers can label secret data with `@Secret` annotations and control the release of secret information to the non-enclave environment via the `declassify` operator as well as the influence of untrusted information from the non-enclave environment via `endorse` operator. J_E builds on the Java information flow (Jif) security-typed language [18] to statically check confidentiality and integrity policies of code running within an enclave. A J_E program is automatically translated into an equivalent Jif program.

To provide security against **realistic attackers**, our key observation is that a program is secure in the presence of enclave active attackers iff the program does not leak additional sensitive information as compared to executing the program in the presence of a passive attacker. Inspired by the line of works on IFC for distributed applications [19], [20], we propose *robustness*, a semantic characterization of security for enclaves for two realistic active attackers. Robustness captures the interplay between the integrity of untrusted data coming from the non-enclave environment and the confidentiality of secret data within the enclave, ensuring that an active attacker does not learn additional information. We enforce robustness with a security type system for a core of J_E to check that the active attackers' control over inputs at the enclave interface and over the execution order of interface components does not enable them to learn more sensitive information than a passive attacker who merely observes the outputs at the interface. Importantly, our security type system can be used to check robustness for the partitioned programs leveraging existing verification efforts for the original program before partitioning. In contrast to existing type systems for robust declassification [19], [20], [21], our security type system is flow sensitive [22] which poses additional challenges with declassification policies and enclave attackers.

To validate the design of J_E, we verify several case studies from the literature. To validate our approach to security for enclaves, we prove our security type system sound with respect to robustness, showing that it accepts only secure programs. The implementation of J_E and the case studies discussed in the paper are publicly available [1]. In summary, this paper makes the following contributions:

- We present J_E, a programming model that seamlessly supports computations inside an enclave. In this model, programmers use annotations to identify the computations to be executed inside the enclave.
- We provide a core calculus for J_E and propose a semantic security model to capture the essence of information flows in the presence of active enclave attackers.
- We enforce robustness statically via a security type system, which we prove sound for programs implemented in J_E.
- We evaluate the applicability of J_E using different case studies.

The paper is structured as follows. Section II introduces enclave technology and the different approaches for enclave software development. Section III overviews the J_E design. In Section IV, we present a security framework for J_E. Section VI describes the implementation. The evaluation is in Section VII. Section VIII presents related work and Section IX concludes.

II. Trusted Execution Environments in a Nutshell

In this section, we introduce TEEs and we refer to Intel SGX, as a concrete implementation. TEEs make use of dedicated processor instructions and modified memory access mechanisms to enable private computations. The principle behind a TEE is to provide applications with memory isolation. With the help of the dedicated instructions, applications can create private memory regions known as *enclaves*. An enclave is essentially a reserved area in the system memory protected by the CPU. Data inside an enclave is protected from high privilege software such as OS, VMM and BIOS. This design leads to two distinct execution environments – the enclave environment (trusted environment) and the non-enclave environment (regular system memory).

Intel Software Guard Extensions (SGX) [1], [2] is an enclave implementation from Intel introduced with the sixth generation Intel core and Intel Xeon E3 v6 server processors.

A. The SGX Mechanism

In SGX enabled systems, the BIOS reserves a contiguous part of the DRAM as Processor Reserved Memory (PRM). Currently, the size of the PRM is limited to 128 MB. Out of this PRM, a memory region of about 90 MB is used as Enclave Page Cache (EPC) which stores the enclave code and data. The content of the enclave memory is encrypted using the Memory Encryption Engine (MEE) and is decrypted only when inside the CPU. This solution protects the enclave data from an attacker having physical access to DRAM. An application creates a new enclave using the `ECREATE` instruction. The enclave memory can only be accessed from inside the enclave. The CPU rejects any attempt to access the enclave memory from the non trusted environment. SGX provides a Remote Attestation (RA) service to verify the integrity of the data loaded inside the enclave. We refer to Costan and Devadas [23] for a detailed description of SGX.

B. Software Development with SGX

There are two main development methodologies used for programming with SGX enclaves. One is to use the C/C++ interface provided by Intel along with the Microsoft Visual Studio IDE. In this approach, a programmer writes their application in C/C++, programs to be executed inside and outside the enclave are written in two separate projects and

[1] https://prg-grp.github.io/je-lang

together they form a complete application. The application is compiled using the Intel compiler integrated into the IDE. The benefit of this approach is that the developer can keep the trusted code base (TCB) minimal. Unfortunately, no such support is available for applications written in managed languages. The other approach is to use systems based on library OSes [24], [25], [26], [27], [28]. They allow running unmodified applications inside the enclave. The application and the library OS are compiled together into an image file and the host OS runs the image inside the enclave.

C. Attacker models with enclaves

In this work, we consider two attacker models: a **passive** attacker and (two variants of) an **active** attacker. A passive attacker cannot modify the execution state of the program (inside or outside the enclave) and can only observe the entire execution state of the non-enclave environment. The rationale is that a developer can partition the original program into an enclave component and a non-enclave component, assuming that the partitioned program will be as secure as the original program, so long as the enclave component contains the secret data as well as any code that accesses these data.

The active attacker model extends the threat model considered in [15], [16] and considers that an active attacker can arbitrarily alter the state of the non-enclave environment and hence it can modify the data exchanged between the two environments. An active attacker can influence the execution of the enclave component either by compromising the integrity of inputs at the interface or by controlling the execution order of the interface components. Both cases may lead to triggering execution paths and side effects that were not possible in the original program, thus enabling learning more sensitive information than a passive attacker.

For both attacker models, we only assume that the enclave hardware and software, except for the J_E source program, are trusted. The J_E uses static analysis to check that a passive attacker learns no more information about secret data than allowed by the original program, and to check that an active attacker learns no more information than a passive attacker. Denial-of-service attacks, arbitrary destruction of enclaves by the host OS, hardware side-channel attacks, and power analysis attacks are out of the scope of this paper. We also ignore covert channels including termination and timing. While termination channels have well-understood information leakage bounds [29], orthogonal techniques such as constant-time programming [30] and predictive mitigation mechanisms [31] can help mitigating some of these concerns.

III. J_E Design

We propose a language design that supports computations inside an SGX enclave. Our approach, J_E, relieves the programmer from dealing with the lower-level details of SGX, such as enclave creation, initialization, and destruction. Moreover, J_E supports security annotations to define application-level policies that are subsequently interpreted as information flow policies and are verified via a security type system.

Listing 1: Language constructs

```
1  @Enclave
2  class Encrypter {
3
4     @Secret static String key;
5
6     @Gateway
7     public static String encrypt(String plaintext) {
8        String cipher = encode(plaintext, key);
9        return declassify(cipher);
10 } }
```

Listing 2: The Endorse operator

```
1  @Enclave
2  class Encrypter {
3
4     @Secret static String key;
5
6     @Gateway
7     public static String encrypt(String plaintext) {
8        String cipher = "";
9        String plaintextE = endorse(plaintext);
10       if (plaintextE.length() == 8) {
11          String var1 = encode(plaintextE, key);
12          cipher = declassify(var1);
13       }
14       return cipher;
15 } }
```

A. J_E Annotations

J_E extends the Java language with annotations to specify the parts of the program that run inside SGX as well as the sensitive data. We demonstrate these features using a routine to securely encrypt data (Listing 1). Encryption is based on a key that should be kept secret; hence the whole encryption operation should run within the enclave. J_E supports the following annotations to define security policies.

a) Class annotation `@Enclave`: A programmer can annotate any class with the `@Enclave` annotation. Hereafter, we refer to such classes as enclave classes. The code and the data that belong to enclave classes are placed inside the enclave. To ensure that data and computations related to encryption take place within the enclave, the class `Encrypter` in Listing 1 is annotated with the `@Enclave` annotation.

b) Field annotation `@Secret`: A programmer can annotate fields of an enclave class with the `@Secret` annotation – we refer to such fields as *secret* fields. The annotation denotes that the field holds sensitive information. Any program construct influenced by a secret field will also be considered as secret in order to prevent flows of any sensitive data from the enclave to the non-enclave environment. In Listing 1, the `key` field is used as a key for encryption. The `key` field is annotated with the `@Secret` annotation to denote that its value must not be leaked to the non-enclave environment.

c) Method annotation `@Gateway`: Static methods defined inside an enclave class can be annotated with the `@Gateway` annotation. We refer to these methods as *gateway* methods. These methods act as an interface between the enclave and the non-enclave environment: the execution switches from the non-enclave environment to the enclave when a gateway method is called from the non-enclave environment. In Listing 1, the `encrypt` method is annotated with the `@Gateway` annotation to ensure that encryption is performed within the enclave. The `encrypt` method encrypts the `plaintext` argument with the

secret key and returns the corresponding cipher text to the external environment. The return value of a gateway method should not be influenced by secret information.

d) Operator `declassify`: The `declassify` unary operator downgrades a secret value into a public value. In Listing 1, the value in `cipher` (Line 8) is influenced by the secret field `key`. Hence, `cipher` is considered sensitive and cannot be returned to the non-enclave environment. However, the encryption can be considered secure as an observer cannot learn useful information by observing only the ciphertext. The `declassify` operator is used to control the release of sensitive information by explicitly declassifying a secret value provided it can be trusted (see the following paragraph).

e) Operator `endorse`: The `endorse` operator endorses an untrusted value into a trusted one. A declassification can get triggered based on the specific value of a variable. As a result, a malicious user can influence the value of a variable to trigger the declassification. J_E ensures that only trusted values are declassified and application of the `declassify` operator does not depend on untrusted values. By default, the arguments to the gateway methods are considered untrusted as they are received from the non-enclave environment. In Listing 2, the gateway method `encrypt` accepts the `plaintext` argument from the non-enclave environment. We explicitly trust the identifier `plaintext` (Line 9). As a result, `plaintextE` is a trusted version of `plaintext` and the declassification no longer depends on an untrusted variable. The program in Listing 2 is valid because the `var1` identifier is declassified (Line 12) and it is trusted. For this reason, the application of the `declassify` operator (Line 12) does not depend on an untrusted value. Also, the code in Listing 1 is invalid as the identifier `cipher` being declassified (Line 8) is untrusted.

B. Compilation phases

The first step of the compilation process involves automatically translating a J_E program into an equivalent Jif program. The generated program is verified using the Jif compiler. If the compilation succeeds, the code for data exchange between the enclave and non-enclave environments along with the initialization code is added. Finally, two JARs corresponding to each environment are generated. A detailed description of the compilation steps is provided in the technical report [32].

C. Execution Model

We execute J_E programs in two separate JVMs, one running in the external environment and one within the enclave. By default, a J_E program starts its execution inside the non-enclave environment. When a program running in the non-enclave environment calls a gateway method, the corresponding parameters are passed to the enclave-environment and the called gateway method executes inside the enclave.

When a gateway method is called from the non-enclave environment, the corresponding arguments are serialized and copied (deep copy) into the enclave. Hence, this semantics results in a distinct copy of each argument inside the enclave. To avoid inconsistent copies of an object in the two environments, once a gateway method is called, we prevent any further usage of all its arguments in the non-enclave environment.

$$
\begin{array}{rcl}
CL & ::= & class\ C\ \{\overline{f}, \overline{m}\} \mid classEnclave\ C\ \{\overline{f}, \overline{m}\} \\
m & ::= & method_{\phi}(\overline{p})\{S; return(e)\} \\
S & ::= & skip \mid if\ e\ then\ S_1\ else\ S_2 \mid S_1; S_2 \mid while\ e\ do\ S \\
 & & x := declassify(e) \mid x := e_G \mid C.f := e_G \\
e & ::= & x \mid v \mid e.f \mid e_1 \oplus e_2 \\
e_G & ::= & e \mid e.m(\overline{p}) \\
v & ::= & n \mid C \mid unit
\end{array}
$$

Figure 1: Syntax of J_E

IV. Security Framework

This section presents a security framework for reasoning about the confidentiality and integrity properties of enclave programs. We formalize a core of J_E by defining the syntax and semantics (Section IV-A and IV-B). Drawing on IFC, we present a security model (Section IV-C) with the following ingredients: a security policy specifying the parts of the program that contain secret/trusted information, and the parts that contain public/untrusted information (Section IV-C1); an attacker model specifying the capabilities of active and passive attackers (Section IV-C2); a security condition capturing a semantic characterization of security with respect to the program semantics, the attacker model, and the security policy. Finally, we present a security type system that enforces the security condition in a sound manner (Section IV-D). The full proofs of our technical results can be found in the technical report [32].

A. J_E Syntax

J_E is an imperative language extended with constructs for static classes, methods and fields as shown in Figure 1. CL represents the list of class definitions in a program. We write $\overline{l}$ for a finite list of elements $l_1, \cdots, l_n$. We distinguish two types of classes: *normal* classes ($class$) which are executed in the non-enclave environment and *enclave* classes ($classEnclave$) which are executed in the enclave environment. Each class is defined with a list of fields $\overline{f}$ and methods $\overline{m}$. We assume classes $C \in Class$, methods $m \in Method$, and fields $f \in Field$ are uniquely identified.

We define a method by the list of formal parameters $\overline{p}$ and method body S, and use the annotation $\phi \in \{G, NG\}$ to indicate whether (G) or not (NG) a method is a gateway. The method body S is a sequence of commands that is executed when the method is called, followed by $return(e)$ for the return value of the method. J_E distinguishes two types of expressions, side-effect free expressions e including variables $x \in Vars$, values $v \in Val$, fields accesses $e.f$, and binary operations $\oplus$, and side-effectful expressions e_G which extend e with method calls $e.m(\overline{p})$. Commands S include standard features such as assignment to class fields and variables, conditionals, loops, and sequencing.

Command $x := declassify(e)$ is semantically equivalent to an assignment and it is used for declassification. Command $x := e_G$, assigns the result of evaluating e_G to the variable x. If e_G is a method call, e.g., the assignment $x := C.m(\overline{p})$ denotes a call to method m of class C with actual parameters $\overline{p}$, and stores the return value in variable x. Similarly $C.f := e_G$ assigns the result of evaluating e_G to the field f of class C. Values are integers $n \in \mathbb{Z}$, class identifiers or unit.

B. J_E Operational Semantics

We define the (big-step) operational semantics of J_E. A configuration $\langle S, \mathcal{M}, \mathcal{H}\rangle$ consists of a command S, a memory $\mathcal{M} = Vars \to Val$ mapping variables to values, and a heap $\mathcal{H} = Class \times Field \to Val$ mapping class and field identifiers to values. A state $\sigma = \langle \mathcal{M}, \mathcal{H}\rangle$ is a pair of a memory $\mathcal{M}$ and a heap $\mathcal{H}$. An observation trace t is a (possibly empty) sequence of events $\beta \in Val$, and $t_1.t_2$ denotes trace concatenation.

We use judgments of the form $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow_t \mathcal{M}', \mathcal{H}'$ to denote that a configuration $\langle S, \mathcal{M}, \mathcal{H}\rangle$ evaluates to memory $\mathcal{M}'$ and heap $\mathcal{H}'$ in *execution mode* $\alpha \in \{N, E\}$ with *static mode* $\delta = Class \cup Var \to \{N, E\}$, and produces an observation trace t. We write $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}', \mathcal{H}'$ if the observation trace is empty and $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow_t \_$ to ignore the final state. We use modes to distinguish between the enclave environment (E) and the non-enclave environment (N). An execution mode α indicates the current execution environment of a configuration, while a static mode δ associates a class identifier or a variable with the execution environment it was assigned to, statically. Abusing notation, we write $\alpha \vdash_\delta \langle S; return(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}, \mathcal{H} \triangleright v$ for commands that yield a value v, via the $return$ command. To simplify the presentation, we assume that the sets of variable, field, and class identifiers of the enclave environment and non-enclave environment are disjoint. An initial configuration starts in the non-enclave execution mode and it executes a sequence of commands S_0 from an initial state $\langle \mathcal{M}_0, \mathcal{H}_0\rangle$, i.e., $N \vdash_\delta \langle S_0, \mathcal{M}_0, \mathcal{H}_0\rangle$.

The semantics of expressions, Figure 2, is mostly standard. We use judgments of the form $\alpha \vdash_\delta \langle e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle$ to denote that an expression e_G evaluates to value v, memory $\mathcal{M}'$, and heap $\mathcal{H}'$ in state $\langle \mathcal{M}, \mathcal{H}\rangle$, execution mode α, static mode δ, and it emits the event β. Rule METHOD interacts with the execution semantics of commands to execute the method body, hence it can potentially alter the execution state. We use the auxiliary functions $getMethod : Class \times Method \to m$ to extract a method definition, and $fields : Class \to \wp(Field)$ extract the set of fields of class C. We write $\mathcal{M}[x \mapsto v]$ to denote a memory $\mathcal{M}$ with variable x assigned the value v. Similarly, $\mathcal{H}[(C, f) \mapsto v]$ denotes a heap $\mathcal{H}$ with field f of class C assigned the value v. The full semantics of expressions is reported in the technical report [32].

Rule FIELD ACCESS evaluates expression e to a class identifier C, ensures that the execution mode and static mode are the same, and performs a lookup of its field f in the heap $\mathcal{H}$. It then returns the resulting value v as well as the memory and heap, which are unchanged.

Rule METHOD evaluates expression e to a class identifier C and uses the $getMethod$ function to get the definition of its method m. Next, it checks that either the execution mode and static mode are the same ($\alpha = \delta(C)$) or that the execution mode is non-enclave (N), the static mode is enclave (E), and m is a gateway method ($\alpha = N \wedge \delta(C) = E \wedge \phi = G$). This condition ensures that only methods from the classes that have the same static mode as the current execution mode can be called, with the exception that the non-enclave execution mode can call the gateway methods of enclave classes. For the latter, our rules enforce the copy semantics in a call-by-value fashion. The rule substitutes the actual parameters for the formal parameters and executes the method body, returning a value. Finally, if the method is a gateway call, the emitted event is the return value v, otherwise the empty event ϵ.

Figure 3 provides an excerpt of evaluation rules for J_E commands. The full set of rules can be found in the technical report [32]. Rule STORE updates the field in the heap with the associated value and checks that $\alpha = \delta(C)$ to ensure that execution mode, and the static mode of class C are the same. The event that STORE emits is equal to the event emitted during the evaluation of expression e_G. Rule RETURN evaluates the expression in the current state and returns the associated value.

C. Security Model

1) Security Policy: J_E adopts security labels to specify application-level policies as information flow policies. A security label ℓ is a tuple $\langle \ell_C, \ell_I\rangle$ of a confidentiality label ℓ_C and an integrity label ℓ_I. We use two levels, *Public* ($\mathbb{P}$) and *Secret* ($\mathbb{S}$), for confidentiality, and two levels, *Trusted* ($\mathbb{T}$) and *Untrusted* ($\mathbb{U}$), for integrity. Intuitively, for confidentiality, data from *Secret* sources should not flow to *Public* sinks unless it is explicitly declassified by the developer, and, for integrity, data from *Untrusted* sources should not flow to *Trusted* sinks unless it is explicitly endorsed by the developer. These requirements are reflected by the ordering relation between security labels, namely $\mathbb{P} \sqsubset \mathbb{S}$ and $\mathbb{T} \sqsubset \mathbb{U}$. The product lattice $\mathcal{L}$ lifts the constraints to an ordering relation over label pairs such that $\ell_1 \sqsubseteq \ell_2$ iff $\ell_{1_C} \sqsubseteq \ell_{2_C}$ and $\ell_{1_I} \sqsubseteq \ell_{2_I}$. The lattice $\mathcal{L}$ defines the *join* $\ell_1 \sqcup \ell_2$ and *meet* $\ell_1 \sqcap \ell_2$ operators to compute *least upper bound* and *greatest lower bound* of two labels, respectively. A security policy is then defined by an assignment of security labels to variables, classes and fields of an J_E program.

2) Attacker Model: We use the security labels to define the view of the memory and heap from the attacker's perspective. In our setting, the attacker is at level $\langle \mathbb{P}, \mathbb{U}\rangle$ and it can observe all of initial program state (i.e., the initial memory and the initial heap) that is labeled as public $\mathbb{P}$. This is because an attacker has full control of the non-enclave state and it is allowed to learn any public data of the enclave state. We assume that each variable, class, and field has an associated security label from the lattice $\mathcal{L}$ as defined by a security mapping $\Gamma : (Var \cup Class \cup Field) \to \mathcal{L}$. We then define

$$\textsc{Field Access}\ \frac{\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad \alpha = \delta(C) \qquad v = \mathcal{H}(C, f)}{\alpha \vdash_\delta \langle e.f, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}$$

$$\textsc{Method}\ \frac{\begin{array}{c}\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad method_\phi(\overline{p})\{S; return(e)\} = getMethod(C, m) \\ \Big((\alpha = \delta(C)) \vee \big(\alpha = N \wedge \delta(C) = E \wedge \phi = G\big)\Big) \qquad \mathcal{M}^* = \mathcal{M}[p_i \mapsto \sigma(q_i)] \quad i = 1, ..., |\overline{p}| \qquad \delta(C) \vdash_\delta \langle S; return(e), \mathcal{M}^*, \mathcal{H}\rangle \Downarrow \mathcal{M}^{*'}, \mathcal{H}' \triangleright v \\ \mathcal{M}' = \mathcal{M}^{*'} \setminus [p_i] \quad i = 1, ..., |\overline{p}| \qquad (\alpha = \delta(C) \Rightarrow \beta = \epsilon) \qquad (\alpha = N \wedge \delta(C) = E \wedge \phi = G \Rightarrow \beta = v)\end{array}}{\alpha \vdash_\delta \langle e.m(\overline{q}), \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle}$$

Figure 2: Excerpt of J_E expression semantics

$$\textsc{Store}\ \frac{\alpha \vdash_\delta \langle e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle \qquad \alpha = \delta(C) \qquad \mathcal{H}'' = \mathcal{H}'[(C, f) \mapsto v]}{\alpha \vdash_\delta \langle C.f := e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \mathcal{M}', \mathcal{H}''} \qquad \textsc{Return}\ \frac{\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}{\alpha \vdash_\delta \langle return(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}, \mathcal{H} \triangleright v}$$

Figure 3: Excerpt of J_E command semantics

indistinguishability over pairs of program states for a security mapping Γ and an attacker at security level $A = \langle \mathbb{P}, \mathbb{U}\rangle$.

Definition 1 (State Indistinguishability)**.** *Two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ are indistinguishable for the attacker A (written $\mathcal{M}_1 =_A \mathcal{M}_2$) iff $\forall x.\ \delta(x) = N$, $\mathcal{M}_1(x) = \mathcal{M}_2(x)$, and $\forall x.\ \delta(x) = E$ such that $\Gamma(x) = \langle \mathbb{P}, -\rangle$, $\mathcal{M}_1(x) = \mathcal{M}_2(x)$.*

Two heaps $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable for the attacker A (written $\mathcal{H}_1 =_A \mathcal{H}_2$) iff $\forall C.\ \delta(C) = N, \forall f \in fields(C)$, $\mathcal{H}_1(C, f) = \mathcal{H}_2(C, f)$ and $\forall C.\ \delta(C) = E, \forall f \in fields(C)$ such that $\Gamma(C.f) = \langle \mathbb{P}, -\rangle$, $\mathcal{H}_1(C, f) = \mathcal{H}_2(C, f)$.

Two program states $\sigma_1 = \langle \mathcal{M}_1, \mathcal{H}_1\rangle$ and $\sigma_2 = \langle \mathcal{M}_2, \mathcal{H}_2\rangle$ are indistinguishable for the attacker A (written $\sigma_1 =_A \sigma_2$) iff $\mathcal{M}_1 =_A \mathcal{M}_2$ and $\mathcal{H}_1 =_A \mathcal{H}_2$.

Intuitively, two indistinguishable memories assign the same value to the public variables inside and outside of enclave. Similarly, two indistinguishable heaps assign the same value to all of the fields of *non-enclave classes* and the public fields of *enclave classes*.

As secret values are stored inside the enclave, the only way for an attacker to learn secret information is by observing the return values of gateway methods. In fact, the attacker observations are captured by our semantics in Figure 3 via traces t and events β. Therefore, we can define indistinguishability for program executions by requiring that any two indistinguishable initial states produce the same observation traces. This implies that an attacker cannot discriminate the two initial states, thus it cannot learn secret information from the enclave environment.

Definition 2 (Execution Indistinguishability)**.** *Let S be a J_E program and σ_1 and σ_2 be two initial states such that $\sigma_1 =_A \sigma_2$. Two executions are indistinguishable (written $N \vdash_\delta \langle S, \sigma_1\rangle \approx_A N \vdash_\delta \langle S, \sigma_2\rangle$) if $N \vdash_\delta \langle S, \sigma_1\rangle \Downarrow_{t_1} \_$ and $N \vdash_\delta \langle S, \sigma_2\rangle \Downarrow_{t_2} \_$ then $t_1 = t_2$.*

In line with existing works [29], execution indistinguishability ignores information leaks that are due to program (non) termination. The definition ensures security w.r.t. a passive non-enclave attacker that observes only the results of gateway method calls. In particular, it rejects all programs that leak secret information to the non-enclave environment. As such condition can be restrictive for most practical scenarios (see Section III for examples), developers resort to various forms of declassification operations to release secret information in a controlled manner. We refer to prior works for an overview of various dimensions of declassification [33]. In our setting, declassification can be dangerous as it can be abused by an active attacker to release secret information in a way that it was not intended by the developer [20]. Next, we define the attackers that are relevant in the context of enclave programs.

3) Passive and Active Attackers: We consider three types of attackers: the passive attacker (PA), the *havoc* active attacker (HAA) limited to modifying parameters passed to gateway calls, and the *havoc reordering* active attacker (HRAA) capable of controlling the order and frequency in which gateway methods are called. Both active attackers are reasonable in our context as the attacker fully controls the non-enclave environment. We use the PA attacker as a reference model to show that the HAA attacker and the HRAA attacker do not learn more secret information than the PA attacker.

The PA does not intervene in the execution of the program, it just observes observation traces (as in Figure 3) to learn secret information from the enclave. The active attackers can influence the execution. The HAA attacker can modify the non-enclave state and hence the parameters passed to gateway methods, thus modifying the behavior of the program executing inside the enclave. This may change the behavior of the program in a way that leaks information about the enclave secrets, e.g., by abusing declassification operations that were intended in the context of a PA attacker. We illustrate the issue via an example inspired by Askarov and Myers [21].

The program in Listing 3 contains a gateway method foo whose return value depends on the $time$ parameter. The intention of the developer, who assumes a PA attacker, is to declassify $secretVal$ only after the $releaseTime$ has elapsed. This is implemented by comparing the $time$ in which method call was issued with the predefined $releaseTime$ (Line 10). Yet, an HAA attacker can arbitrarily change the value of $time$ and control the release of information via declassification.

Listing 3: HAA Attacker

```
@Enclave
class FooClass {

   @Secret static int secretVal;
   static int releaseTime = 2025

   @Gateway
   public static int foo(int time) {
      int res = 0;
      if (time >= releaseTime)
         res = declassify(secretVal);
      else
         res = 0;

      return res;
} }
```

Listing 4: HRAA Attacker

```
@Enclave
class FooClass {

   @Secret static int secret1, secret2;
   static boolean releaseTrigger = false;

   @Gateway
   public static void bar() {
      releaseTrigger = true;
   }
   @Gateway
   public static int foo() {
      int res = 0;
      if (releaseTrigger) {
         releaseTrigger = !releaseTrigger;
         res = declassify(secret1); }
      else {
         releaseTrigger = !releaseTrigger;
         res = declassify(secret2); }

      return res;
} }
```

Listing 5: Delayed Declassification

```
@Enclave
class FooClass {

   @Secret static int secretVal;

   @Gateway
   public static int foo(int input) {
      int x = declassify(secretVal);
      int l = 0;
      if (input > 0)
         l = x;
      else
         l = 7

      return l;
} }
```

This example motivates the need for a security condition that rejects scenarios where an active attacker learns more secret information than a passive attacker.

To model the active capability of the HAA attacker, we introduce program holes $[\bullet]$ [20]. Holes represent program contexts where an HAA attacker can insert an untrusted code a to modify the program's state. Because the only way an attacker can affect the execution of enclave code is via gateways, we extend our program syntax by adding $[\bullet]; x := e.m(\overline{p})$ and $[\bullet]; C.f := e.m(\overline{p})$, and define the active attacks.

Definition 3 (J_E program with holes)**.** *A program with holes $S[\overrightarrow{\bullet}]$ is defined by extending the syntax in Figure 1 as follows:*

$$S[\overrightarrow{\bullet}] ::= \ldots \mid [\bullet]; x := e.m(\overline{p}) \mid [\bullet]; C.f := e.m(\overline{p})$$

where m is a gateway method.

An HAA attacker can execute any untrusted code a in a hole before method calls. However, this code can only contain variables, classes, and fields in the non-enclave environment which are by definition public and untrusted. We define the attacker's code as follows:

$$a ::= skip \mid a_1; a_2 \mid q := e \ (where\ q \in \overline{p}) \qquad (1)$$

While an HAA attacker can inject arbitrary untrusted code in the non-enclave environment, we argue that the definition above captures the most powerful attack strategies of HAA attacker.

Lemma 1. *The attack code definition presented in 1, captures the most powerful attack strategies available to an HAA attacker, who controls the non-enclave environment.*

We write $S[\overrightarrow{a}]$ for a program under an HAA attack $\overrightarrow{a}$. In this setting, a passive attack can be modeled as $S[\overrightarrow{skip}]$.

An HRAA attacker controls both the code and data memory outside the enclave. Hence in addition to the HAA attack capability, an HRAA attacker can change the *order* and *frequency* of gateway method calls issued from the non-enclave environment. This can cause information leaks as the (attacker-controlled) order and frequency of gateway calls may influence the values returned to the non-enclave environment. Listing 4 illustrates the problem. Consider the non-enclave program `FooClass.bar(); FooClass.foo()`. The intended order of issuing gateway methods is `bar();foo()`, hence this program is secure with respect to an HAA attacker. `secret1` is always going to be declassified, and the resulting trace depends on its value. However, an HRAA attacker that controls the code memory outside the enclave can change the order of gateway calls to `foo();bar()` and learn the declassified value of `secret2`.

A similar argument applies to calling gateway methods multiple times. For instance, if a gateway was intended to be called only once, calling it more than once might leak sensitive information. We revisit Listing 4 to illustrate the problem. Consider the non-enclave program `FooClass.foo()` revealing the value of `secret1`. An HRAA attacker can instead call `FooClass.foo();FooClass.foo()` and learn the declassified value of `secret2`.

Since the HRAA attacker has full control over the non-enclave code and memory, and the secrets reside only inside the enclave, we model them as sequences of gateways calls.

Definition 4 (Program under HRAA control)**.** *We define the program under HRAA control as a sequence of gateway calls:*

$$S'[\overrightarrow{\bullet}] ::= S'_1[\overrightarrow{\bullet}]; S'_2[\overrightarrow{\bullet}] \mid [\bullet]; x := C.m(\overline{p})$$

where m is a gateway method defined in $S[\overrightarrow{\bullet}]$.

In this model, the attack definition of 1 will remain unchanged, indicating that HRAA attacker subsumes the power of HAA attacker.

Lemma 2. *The attacker code defined in 1, captures the most powerful attack strategies available to the HRAA attacker.*

4) Security Condition: In our setting, programs use declassification to release secret information in a controlled manner. Intuitively, a program is secure if an active attacker cannot learn more secret information than a passive attacker. Drawing on the idea of robust declassification [20], we present

robustness, a security condition that formalizes this intuition.

Definition 5 (Robustness under HAA)**.** *Program $S[\overrightarrow{\bullet}]$ is robust w.r.t an HAA attacker A if for all $\sigma_1, \sigma_2, \overrightarrow{a}_1, \overrightarrow{a}_2$*

$$N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_2\rangle \Rightarrow N \vdash_\delta \langle S[\overrightarrow{a}_2], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_2], \sigma_2\rangle$$

Robustness holds whenever for an attack vector $\overrightarrow{a}$ and two indistinguishable initial states σ_1, σ_2, if the program $S[\overrightarrow{a}_1]$ satisfies execution indistinguishability (definition 2), then for another attack vector $\overrightarrow{a}_2$, the program $S[\overrightarrow{a}_2]$ also satisfies execution indistinguishability. In other words, the attacker observations of $S[\overrightarrow{a}_2]$'s observation traces do not reveal any secrets apart from what the attacker already knows by the observation traces of the program $S[\overrightarrow{a}_1]$. The PA attacker is captured by executions of the program $S[\overrightarrow{skip}]$.

We extend robustness to capture HRAA attackers.

Definition 6 (Robustness under HRAA)**.** *Program $S[\overrightarrow{\bullet}]$ is robust w.r.t an HRAA attacker A if for all $\sigma_1, \sigma_2, \overrightarrow{a}_1, \overrightarrow{a}_2$ and for all $S'[\overrightarrow{\bullet}]$:*

$$N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_2\rangle \Rightarrow N \vdash_\delta \langle S'[\overrightarrow{a}_2], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a}_2], \sigma_2\rangle$$

This definition ensures that the HRAA attacker does not learn more information by changing the order and frequency of gateway calls. Observe that if $S'[\overrightarrow{a'}]$ triggers an execution that was not possible $S[\overrightarrow{a}]$, and that execution's return value depended on some declassified secret, the active power may enable the HRAA attacker to learn information that they would not have learned originally. In Listing 3, the program $S[\overrightarrow{a}] ::= [a_1]; FooClass.bar(); [a_2]; FooClass.foo()$ is not robust under HRAA because the program $S'[\overrightarrow{a'}] ::= [a'_1]; FooClass.foo(); [a'_2]; FooClass.bar()$ enables the HRAA attacker to reveal the declassified value of `secret2`. A similar argument applies to calling gateway methods that were defined in $S[\overrightarrow{\bullet}]$ but were never called. As expected, this definition is stronger than Definition 5.

5) Delayed Declassification: While our security condition extends the definition of robust declassification [20] to the setting of realistic enclave attackers, there are some key differences pertaining to traces and attacker observations. Robust declassification considers every assignment to public variables as immediately visible to the attacker, because it defines the observations as projection over the public part of the memory. This definition does not reflect the enclave attacker model, because TEEs encrypt the entire enclave memory, so even if a public variable is modified inside the enclave, it is not visible to the attacker unless it is written to the non-enclave memory. This motivates our use of a trace-based observation model as generated by the return values of gateway method calls.

This model poses additional challenges with handling of declassification policies. For example, the program in Listing 5 satisfies robust declassification because the attacker $input$ neither affects the decision to declassify nor the declassified value itself. This is achieved by making the value in variable x observable to the attacker immediately. However, in our model, the declassified value in x will not be visible to the attacker until it is returned by the gateway method. Because the attacker controls the input and therefore the assignment in line 11, this results in controlling the decision to declassify the secret value. Hence, the program is not robust. In fact, for attack vectors $a ::= input := 0$ and $a' ::= input := 1$, our definition of robustness will correctly reject the program. We dub this concept *delayed declassification*. Observe that delayed declassification is orthogonal to the well-known *Where* and *What* dimensions of declassification [33] and it appears as result of the trace-based observation model.

D. Security Type System

This section presents a security type system to enforce robustness. In line with the enclave attacker model, the program from the non-enclave environment is public and untrusted, while program from the enclave environment is trusted and its input data can be labeled by the developer as either public or secret. We label secret fields with $\langle \mathbb{S}, \mathbb{T}\rangle$ as they contain sensitive information and are protected by the enclave. The security label of the arguments of a gateway method is defaulted to $\langle \mathbb{P}, \mathbb{U}\rangle$ and while the return parameter is labeled as $\langle \mathbb{P}, \mathbb{T}\rangle$ or $\langle \mathbb{P}, \mathbb{U}\rangle$.

The goal of our type system is to ensure robustness against the active attackers. To achieve this, the type system checks that the decision to declassify a secret, or to return it, is not influenced by untrusted non-enclave data, thus ensuring that the attacker cannot control the decision to release secret data. The security type system enforces the *Where* dimension of declassification for a PA attacker [33] and it has been proved sound with respect to the security condition of *gradual release* [34]. We extend the type system to additionally account for active attacks in our setting and prove it sound for robustness. This has the advantage of reusing existing verification efforts via security type systems, which assume a PA attacker, and verifying only on the effect of an active attacker whenever these programs have been verified for the PA attacker. In our setting, this may happen whenever a developer partitions an existing (secure) program to execute with enclaves.

Our security type system uses a typing environment $\Gamma : (Var \cup Class \cup Field) \to \mathcal{L}$ mapping variables, classes, and fields to security labels from the security lattice $\mathcal{L}$. We also use another environment $\Pi : (Var \cup Class \cup Field) \to \mathcal{B}$ that maps variables, fields, and classes to boolean flags. A flag $d \in \mathcal{B}$ can be true (T) or false (F), and is used to track the propagation of declassified values. Initially, every variable and field has the flag initialized to false and the flag is set to true whenever they store a value that may be affected by declassification. We define the ordering relation $F \sqsubset T$ on flags, which will be useful in the sub-typing and method rules.

The label of every variable, field, and class in our setting, is a tuple consisting of a security label and a flag (ℓ, d). We use indexes ℓ and d to access elements of this tuple. e.g., pc_ℓ will

show the security label of pc. Methods are typed in isolation using type signatures of the form $\left(\Gamma^-, \Pi^- \xrightarrow[pc']{} \Gamma^+, \Pi^+\right)_{(\ell,d)}$ which require a environments Γ^- and Π^- before the method is invoked, environments Γ^+ and Π^+ after the method invocation, the label of its return value (ℓ, d), and the program counter label pc' capturing the lower bound on method's side-effects [16]. The typing judgments for expressions are of the form $\Gamma, \Pi \vdash_\delta e_G : \tau$, meaning that in mode δ, and environments Γ and Π, an expression e_G has the type τ. If the expression e_G is a method, then τ is a method type, otherwise τ is a type (ℓ, d) representing the security label of e. Similarly, the typing judgments for commands have the form $pc, \Gamma, \Pi \vdash_\delta S : \Gamma', \Pi'$ where Γ and Π are the environments before, and Γ' and Π' are the environments after the execution of command S, δ is the static, and pc is the program counter label used to prevent implicit flows. The judgment for the return command is of the form $pc, \Gamma, \Pi \vdash_\delta return(e) : \Gamma, \Pi \triangleright (\ell, d)$ to capture the security label ℓ and the flag d of the returned value.

Figure 4 depicts an excerpt of the typing rules for expressions. The full list of rules is reported in the technical report [32]. The only non-standard rule here is T-METHOD. Rule T-METHOD checks that method's body $S; return(e)$ is well-typed under pc', Γ^-, Γ^+, Π^-, and Π^+ and it returns the label (ℓ, d) of the result.

Figure 5 presents a few interesting rules for commands; we refer to the technical report [32] for the full list. In our type system, variables are flow-sensitive (see rule T-ASSIGN in the technical report [32]), while fields are flow-insensitive, i.e., their security label is defined via annotations. In fact, rule T-STORE ensures that the join of the security labels of pc and expression e is at least as restrictive as the label of field $C.f$. Moreover, the rule ensures that if the security label of $\ell_1 \sqcup \ell_2 \sqcup pc_\ell$ is secret $\langle \mathbb{S}, - \rangle$, the store is defined in the enclave, and it ensures that if the flag of e is true, this command can only be executed in a trusted context. This is to prevent untrusted input from controlling the propagation and release of declassified values. Finally, even though $C.f$'s security labels are flow-insensitive, its flag is not, and it is updated to the disjunction of the flags of e and pc. This rule also updates the flag of class C, so if a class has a field affected by a declassified value, the whole class is going to be flagged true. The rule T-STORE is only for side-effect free expressions (i.e., e). There is also $e.m(\overline{p})$ that combined with Store can act as a method call or a *gateway* method call. Rule T-STORE-CALL and T-STORE-GATEWAY-CALL type check method calls and gateway method calls, respectively. Similarly, type checking assignments is broken into three rules, T-ASSIGN, T-ASSIGN-CALL, and T-ASSIGN-GATEWAY-CALL.

Rule T-DECLASSIFY ensures that only trusted data is allowed to be declassified $\ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle$ and declassification can only happen in public and trusted context $pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle$. This prevents attacker-controlled untrusted data from influencing the decision to declassify secret information. The security label of variable x will be $\langle \mathbb{P}, \mathbb{T} \rangle$ and its flag will be true T.

Rule T-RETURN sets the security label of the returned value to the join of the security labels of program context and the expression. This is to prevent the implicit flows that may happen when returning in a secret context. Additionally, if the expression e's flag is true, return can only be executed in a trusted context. This is to prevent attacker from affecting the decision to return a declassified value.

Rule T-STORE-GATEWAY-CALL handles the type checking of gateway methods call from a store. The security labels of gateway parameters are explicitly defined in Γ^- and must have the $\langle \mathbb{P}, \mathbb{U} \rangle$ security label ($\forall p \in \overline{p}.\ \Gamma^-(p) = \langle \mathbb{P}, \mathbb{U} \rangle$), and their flag (defined in Π^-) should be false $\forall p \in \overline{p}. \Pi^-(p) = F$. The rule ensures that the gateway can only return public values ($\ell_2 \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle$), the labels of actual parameters are less restrictive than the predefined labels of formal parameters $\Gamma(q_i) \sqsubseteq \Gamma^-(p_i)$, and the method's typing environment is satisfied ($\forall y \in dom(\Gamma^-).\Gamma(y) \sqsubseteq \Gamma^-(y)$). Additionally, the typing environment after type checking the method body should respect the method's predefined post typing environment ($\forall y \in dom(\Gamma^+).\Gamma^+(y) \sqsubseteq \Gamma_{out}(y)$), while ensuring that the type of identifiers other than those used by the method remains unchanged ($\forall y \in dom(\Gamma) \setminus dom(\Gamma^+).\Gamma(y) = \Gamma_{out}(y)$). Similar conditions apply to Π environment. At last, the rule updates the flag of field $C.f$ to false, so even a declassified value, after returning from a gateway method has a false flag.

To illustrate our type system, we revisit the example of Listing 3. The type system rejects this program because the declassify operator can only be used in a trusted program context pc. Rules T-OP and T-IF-ELSE set the pc security label to the join of the security labels of $time$ and $releaseTime$. Since $time$ has security label $\langle \mathbb{P}, \mathbb{U} \rangle$ (as it comes from outside of the enclave), and $releaseTime$ has security label $\langle \mathbb{P}, \mathbb{T} \rangle$ (as it comes from the enclave), we have that $pc_\ell = \langle \mathbb{P}, \mathbb{U} \rangle \sqcup \langle \mathbb{P}, \mathbb{T} \rangle = \langle \mathbb{P}, \mathbb{U} \rangle$. Next, rule T-DECLASSIFY does not allow the declassification since $pc_\ell \not\sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle$.

We prove that our security type system enforces robustness for the HAA attacker.

Theorem 1. *If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HAA.*

We now show how it can be extended with minimal changes to enforce robustness for HRAA attackers. The additional power of HRAA attacks comes from the ability to control the order and frequency of gateway calls. Whenever a gateway call modifies trusted identifiers inside the enclave, this can be used to influence declassification operations that are performed by another method. This can be prevented by computing the set of *shared identifiers* Σ containing all global variables and fields that are *assigned* to in at least one method and *accessed* in at least one method. (i.e., Σ contains all global variables and fields that are used and modified in one or more gateways methods) We compute Σ in a preprocessing step up to reaching a fixed point. We can then use the typing rules of Figures 4 and 5 under the constraint that the initial environment considers the integrity label of every variable and field in Σ as *untrusted*. i.e., $\forall\ x \in \Sigma.\ \Gamma_0[x \mapsto \langle \Gamma_0(x)_C, \mathbb{U} \rangle]$ and $\forall\ C.f \in \Sigma.\ \Gamma_0[C.f \mapsto \langle \Gamma_0(C.f)_C, \mathbb{U} \rangle]$.

$$\text{T-INT}\ \frac{}{\Gamma, \Pi \vdash_\delta n : (\langle \mathbb{P}, \mathbb{T} \rangle, F)} \qquad \text{T-OP}\ \frac{\Gamma, \Pi \vdash_\delta e_1 : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta e_2 : (\ell_2, d_2)}{\Gamma, \Pi \vdash_\delta e_1 \oplus e_2 : (\ell_1 \sqcup \ell_2, d_1 \vee d_2)}$$

$$\text{T-METHOD}\ \frac{method_\phi(\overline{p})\{S; return(e)\} = getMethod(C, m) \qquad pc', \Gamma^-, \Pi^- \vdash_\delta S; return(e) : \Gamma^+, \Pi^+ \triangleright (\ell, d)}{\Gamma, \Pi \vdash_\delta C.m(\overline{p}) : \left(\Gamma^-, \Pi^- \xrightarrow[pc']{} \Gamma^+, \Pi^+\right)_{(\ell,d)}}$$

Figure 4: Excerpt of typing rules for expressions

$$\text{T-STORE}\ \frac{\begin{array}{c}\Gamma, \Pi \vdash_\delta C.f : (\ell_1, d_1) \\ \Gamma, \Pi \vdash_\delta e : (\ell_2, d_2) \qquad \ell_2 \sqcup pc_\ell \sqsubseteq \ell_1 \qquad \ell_1 \sqcup \ell_2 \sqcup pc_\ell = \langle \mathbb{S}, - \rangle \Rightarrow \delta(C) = E \qquad d_2 = T \Rightarrow pc_\ell = \langle -, \mathbb{T} \rangle \qquad d' = d_2 \vee pc_d\end{array}}{pc, \Gamma, \Pi \vdash_\delta C.f := e : \Gamma, \Pi\big[C.f \mapsto d', C \mapsto \Pi(C) \vee d'\big]}$$

$$\text{T-DECLASSIFY}\ \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle \qquad pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := declassify(e) : \Gamma\big[x \mapsto \ell \sqcap \langle \mathbb{P}, \mathbb{T} \rangle\big], \Pi\big[x \mapsto T\big]} \qquad \text{T-RETURN}\ \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad d = T \Rightarrow pc_\ell = \langle -, \mathbb{T} \rangle}{pc, \Gamma, \Pi \vdash_\delta return(e) : \Gamma, \Pi \triangleright (pc_\ell \sqcup \ell, d)}$$

$$\text{T-STORE-GATEWAY-CALL}\ \frac{\begin{array}{c}\Gamma, \Pi \vdash_\delta C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta C'.m(\overline{p}) : \left(\Gamma^-, \Pi^- \xrightarrow[pc']{} \Gamma^+, \Pi^+\right)_{(\ell_2, d_2)} \qquad \delta(C) = N \qquad \delta(C') = E \\ \ell_2 \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Gamma^-(p) = \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Pi^-(p) = F \qquad \ell_2 \sqcup pc_\ell \sqsubseteq \ell_1 \qquad \Gamma(q_i) \sqsubseteq \Gamma^-(p_i) \quad i = 1...|\overline{p}| \\ \Pi(q_i) = \Pi^-(p_i) \quad i = 1...|\overline{p}| \qquad \forall y \in dom(\Gamma^-).\Gamma(y) \sqsubseteq \Gamma^-(y) \qquad \forall y \in dom(\Gamma^+).\Gamma^+(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (dom(\Gamma) \setminus dom(\Gamma^+).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in dom(\Pi^-).\Pi(y) \sqsubseteq \Pi^-(y) \\ \forall y \in dom(\Pi^+).\Pi^+(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (dom(\Pi) \setminus dom(\Pi^+).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_\delta C.f := C'.m(\overline{q}) : \Gamma_{out}, \Pi_{out}\big[C.f \mapsto F\big]}$$

Figure 5: Excerpt of typing rules for commands

The intuition is that by considering shared identifiers as untrusted, we enable the security type system to reject programs that use these identifiers to declassify information (explicitly or implicitly) and return it via a gateway method. For example, the program in Listing 4 is rejected by our type system. After the preprocessing phase, $\Sigma = \{releaseTrigger\}$ since $releaseTrigger$ is assigned to in one method call and accessed in another. By assigning the security label $\langle \mathbb{P}, \mathbb{U} \rangle$ to $releaseTrigger$, rule T-DECLASSIFY (Line 16) will fail since declassification is not allowed in an untrusted context.

Recall from last section that if different program executions call different gateways and those gateways return *declassified values*, the HRAA attacker can learn more by calling the gateways that were not called in the original program. In order to prevent this, we add another step to the type checking process to ensure that all gateways which declassify secret values are called in all possible executions of the (non-enclave) program.

This process is performed in several steps:

1) Identify the set of all the *gateway* methods in $S[\overrightarrow{\bullet}]$ denoted by G^D such that their return value's flag is true (i.e., $\Gamma, \Pi \vdash_\delta C.m(\overline{p}) : (-)_{(-,T)}$).
2) Enumerate the paths of the program and extract the set of gateway calls along those paths. This is achieved by a depth-first traversal of the program's graph. We use $pathsGW(S[\overrightarrow{\bullet}])$ to denote the set of all possible paths of $S[\overrightarrow{\bullet}]$, and $pathGW_i$ for the set of gateway calls in path i.
3) Check that G^D is a subset of all of the possible paths of $S[\overrightarrow{\bullet}]$. In other words:

$$G^D \subseteq pathGW_i \quad \forall i \in pathsGW(S[\overrightarrow{\bullet}])$$

This process is performed after calculating Σ and type checking methods in isolation, but before type checking the program itself. If the above process fails, we reject the program as not robust against HRAA attacker. This requirement reflects the power of the HRAA attacker outside the enclave, rejecting programs that do not call all declassifying gateways. To improve permissiveness and security, such programs can be moved to the enclave and exposed to the non-enclave environment as a single gateway method.

We remark that our enforcement accepts programs with noninterfering runs that do not always declassify information. Listing 6 presents a program with both interfering and noninterrfering runs. The program satisfies robustness (Definition 6) and is also accepted by the type system. In fact, if variable `trustedLow` (Line 4, a public and trusted variable) is set to true, only noninterfering computations will be executed and an HRAA attacker program such as `[choice=v];ComputeArray.compute(choice);` will observe the result `"Done with computation" + choice`. Otherwise, if `trustedLow` is set to false, an interfering run will be executed and the attacker will observe the declassified average value.

This example demonstrates that even though our type system requires calling all of the declassifying gateways along all program's executions, it does not mean that all of the runs of the program are necessarily interfering.

We prove soundness of the security type system with respect to the HRAA attacker.

Listing 6: Noninterfering Runs under HRAA

```
@Enclave
class ComputeArray {

   static boolean trustedLow;
   @Secret static int[] array;

   @Gateway
   public static String compute(int choice) {
      int avg = computeAvg(array);
      String res = 0;
      if (trustedLow) {
         if (choice == 1) {
            computation1(array); //noninterfering
         } else if (choice == 2) {
            computation2(array); //noninterfering
         } else if (choice == 3) {
            computation3(array); //noninterfering
         }
         res = "Done with computation" + choice;
         }
      }
      else {
         res = String.valueOf(declassify(avg));
      }
      return res;
} }
```

Theorem 2. *If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *with regard to* Σ *and* G^D*, then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HRAA.*

Use case for the HRAA attacker Our enforcement mechanism for the HRAA attacker requires a program to call all gateways that declassify secret information in any execution of a program. While this condition may seem restrictive, it is necessary in order to ensure that an attacker as powerful as HRAA cannot manipulate the program to tamper with declassification in unintended ways.

We identify the setting of IoT app platforms as a promising use case for enforcing security against the HRAA attacker [35]. IoT apps allow users to run simple trigger-action apps in cloud-based IoT platforms to seamlessly connect their IoT services and devices. Upon the triggering of an event, e.g., "EZVIZ camera senses motion at home", the app executes code to perform an action, e.g., "Send an email with the camera stream". Currently, the users have to trust the cloud provider with the sensitive information of their services and devices to run apps on their behalf. TEEs can help executing user apps securely in an untrusted IoT cloud platform and protect against the HRAA attacker. Specifically, the user can leverage enclaves to securely connect their services and smart devices, e.g., Email and EZVIZ camera, via authentication tokens (using code patterns similar to Listing 9) and use these tokens to execute trigger-action automations via gateway methods that simply transfer data between services and devices, and do not declassify sensitive information. In this setting, only the authentication gateway declassifies sensitive information, thus making these programs amenable to verification by our type system.

V. Endorsement and Nonmalleable Attacks

A. Endorsement

We extend our security framework to accommodate explicit endorsement of untrusted information coming from the non-enclave environment via gateway calls. This enables a developer to mark untrusted expressions as trusted, explicitly, to indicate that the security policy should be insensitive to their value. Following the approach of Askarov and Myers [21], we extend the syntax and semantics of J_E with command $x := endorse_\eta(e)$. Each endorsement command has a unique label η, and produces an endorsement event $endorse(\eta, v)$, which records the label η along with the endorsed value v.

$$\textsc{Endorse}\ \frac{E \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle \quad \delta(x) = E \quad \mathcal{M}' = \mathcal{M}[x \mapsto v]}{E \vdash_\delta \langle x := endorse_\eta(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow_{endorse(\eta,v)} \mathcal{M}', \mathcal{H}}$$

We define *irrelevant* attacks as the set of attacks that are endorsed, and therefore can be excluded from the set of attacks used in definition 5. Using this concept, we argue that definition 5 should only hold for relevant attacks. Given a program $S[\overrightarrow{\bullet}]$, starting state σ, and attacker vector $\overrightarrow{a}$ which produces trace t (i.e., $\langle S[\overrightarrow{a}], \sigma\rangle \Downarrow_t$), relevant attacks, denoted by $\Omega(S[\overrightarrow{a}], \sigma)$, are the attacks that lead to the same sequence of endorsement events as in t.

We define relevant attacks by using irrelevant traces, which given trace t, are the set of all traces that agree with t on some prefix of *endorsement events* until they necessarily disagree on some endorsement. Formally:

Definition 7 (Irrelevant Traces)**.** *Given a trace* t*, where endorsements are marked as* $endorse(\eta_j, v_j)$*, define a set of irrelevant traces based on the number of endorsements in* t

$$\psi_i(t) = \{t' \mid t' = k.endorse(\eta_i, v_i').k'\}$$

where k *is a prefix of* t' *with* $i-1$ *events all of which agree with* ***endorse events*** *in* t*, and* $v_i \neq v_i'$*.*

We define $\psi(t) \triangleq \bigcup_i \psi_i(t)$ *as a set of irrelevant traces w.r.t. trace* t*.*

Now, we can define the relevant attacks as the set of attacks that *do not* lead to irrelevant traces.

Definition 8 (Relevant Attacks)**.** *Given a program* $S[\overrightarrow{\bullet}]$*, starting state* σ*, and attacker vector* $\overrightarrow{a}$ *such that* $\langle S[\overrightarrow{a}], \sigma\rangle \Downarrow_t$*, relevant attacks* $\Omega(S[\overrightarrow{a}], \sigma)$ *are defined as:*

$$\Omega(S[\overrightarrow{a}], \sigma) = \{a' \mid \langle S[\overrightarrow{a'}], \sigma\rangle \Downarrow_{t'} \wedge\ t' \notin \psi(t)\}$$

Using the definition relevant attacks, we can redefine the robustness property. This new security condition accounts for the fact that the active effect on endorsed expressions does not matter.

Definition 9 (Robustness with Endorsement)**.** *Program* $S[\overrightarrow{\bullet}]$ *is robust w.r.t an HAA attacker* A *if for all* $\sigma_1, \sigma_2, \overrightarrow{a}_1$ *and for all* $\overrightarrow{a}_2 \in \Omega(S[\overrightarrow{a}_1], \sigma_1)$

$$N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_2\rangle \Rightarrow \\ N \vdash_\delta \langle S[\overrightarrow{a}_2], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_2], \sigma_2\rangle$$

This definition is similar to the original robustness property, except that instead of ensuring indistinguishability for all possible attacks, it only ensures indistinguishability for the

$$\text{T-ENDORSE}\ \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := endorse_\eta(e) : \Gamma\big[x \mapsto \ell \sqcap \langle \mathbb{S}, \mathbb{T} \rangle\big], \Pi\big[x \mapsto d\big]}$$

Figure 6: Typing rule for endorse command

relevant attacks, effectively ignoring the influence of irrelevant attacks.

Figure 6 presents the typing rule for endorsement. Similar to the T-DECLASSIFY, T-ENDORSE ensures that $endorse_\eta(e)$ can only be used inside the enclave environment, and endorsement can only occur in a public and trusted context $pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle$. This is to prevent attacker-controlled untrusted data to influence the decision to endorse untrusted information.

Now, we present type soundness to prove that any well-typed program under the extended type system satisfies the new robustness property of Definition 9.

Theorem 3. *If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ then $S[\overrightarrow{\bullet}]$ satisfies robustness with endorsement.*

We can use a similar approach to model endorsement of HRAA attacker. However, we have to modify the definition of relevant attacks to account for the difference between the order and frequency of endorsement events in $S'[\overrightarrow{\bullet}]$ and $S[\overrightarrow{\bullet}]$.

In the new definition of relevant attacks, we use the unique endorsement label η to ensure that the value of an endorsement used in $S[\overrightarrow{\bullet}]$ is equal to the value of that same endorsement in $S'[\overrightarrow{\bullet}]$ independently of the order and frequency of that endorsement's event. We lift the definition of irrelevant traces and relevant attacks to this new setting.

Definition 10 (HRAA Irrelevant Traces)**.** *Given a trace t, where endorsements are marked as $endorse(\eta_j, v_j)$, define the set of irrelevant traces as:*

$$\psi_R(t) = \{t' \mid \exists\ endorse(\eta_j, v_j) \in t, \exists\ endorse(\eta_i, v_i) \in t'.\ \eta_j = \eta_i\ and\ v_j \neq v_i\}$$

Observe that we need Definition 10 because, if we used Definition 7, there might be cases where there is an irrelevant trace t', but there exist no k with $i-1$ events such that all of its endorse events agree with t, thus, t' cannot be marked as irrelevant.

Definition 11 (HRAA Relevant Attacks)**.** *Given programs $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$, starting state σ, and attacker vector $\overrightarrow{a}$ such that $\langle S[\overrightarrow{a}], \sigma\rangle \Downarrow_t$, the set of relevant attacks w.r.t HRAA attacker $\Omega_R(S[\overrightarrow{a}], S'[\overrightarrow{\bullet}], \sigma)$ are defined as:*

$$\Omega_R(S[\overrightarrow{a}], S'[\overrightarrow{\bullet}], \sigma) = \{a' \mid \langle S'[\overrightarrow{a'}], \sigma\rangle \Downarrow_{t'} \wedge\ t' \notin \psi(t)\}$$

Now, using this new definition of relevant attacks, we can redefine the robustness property w.r.t HRAA attacker.

Definition 12 (Robustness under HRAA with Endorsement)**.** *Program $S[\overrightarrow{\bullet}]$ is robust w.r.t an HAA attacker A if for all $\sigma_1, \sigma_2, \overrightarrow{a}_1$, and for all $S'[\overrightarrow{\bullet}]$ such that $\forall \overrightarrow{a}_2 \in \Omega_R(S[\overrightarrow{a_1}], S'[\overrightarrow{\bullet}], \sigma_1)$*

$$N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}_1], \sigma_2\rangle \Rightarrow N \vdash_\delta \langle S'[\overrightarrow{a}_2], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a}_2], \sigma_2\rangle$$

This definition ensures that in a robust program w.r.t HRAA attacker, as long as the endorsed values are equal, the result of execution for all possible orders and frequencies of gateway calls is indistinguishable.

We can augment the security type system of HRAA attacker with rule of Figure 6 to enforce robustness. The following theorem proves that any well-typed program under the extended type system w.r.t HRAA attacker satisfies the robustness property of Definition 12.

Theorem 4. *If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ with regard to Σ and G^D, then $S[\overrightarrow{\bullet}]$ satisfies robustness under HRAA with endorsement.*

B. Nonmalleable Attacks

Transparent endorsement [36] was introduced as a dual to robust declassification to prevent attacks arising from trusted agents endorsing information that their provider could not have seen. It is a common practice in TEE to have encrypted data passed to the enclave from outside, if these encrypted data are labeled *secret*, then the type system will reject the program and prevent the malleable attacks. Otherwise, if these encrypted ciphertexts are labeled *public* even though they contain secret plaintexts, then it is possible to receive an input which its provider cannot read, thus opening up the system to malleable attacks. While from a technical perspective transparent endorsement can be accommodated into our framework along the lines of Cecchetti et al. [36], we postpone it to future work.

VI. IMPLEMENTATION

In this section, we describe the implementation of J_E for Intel SGX and discuss the gap with our formal model.

The J_E runtime operates with two separate execution environments, namely the non-enclave environment and enclave environment, which correspond to two separate Java Virtual Machines. The communication between the two environments is achieved using Java RMI [37].

The J_E compilation process involves multiple steps. A J_E program is first partitioned, then translated to Jif to check security, and finally transformed to use RMI communication. A detailed description of the code transformations implemented by the J_E compiler with a complete step-by-step example is in the technical report [32].

In addition to the abstractions supported in the formal model, i.e., static classes and static fields, the implementation supports additional Java features, including objects and generics. We convert J_E programs that include objects and generics into equivalent Jif programs. Because the implementation leverages Jif for label propagation, J_E security analysis is flow-insensitive, in contrast to the security type system, where the analysis is flow-sensitive on program variables. In the

Listing 7: Password checker

```
1 @Enclave
2 class PasswordChecker {
3    @Secret static String password;
4
5    @Gateway
6    public static boolean checkPassword(String guess) {
7       return (getHash(guess) == declassify(getHash(password)));
8 } }
```

implementation, we do not assign any default Jif label to the local variables, hence, local variables get the security label of the expression they are initialized with. We remark that this decision is sound with respect to the language subset considered in our formal model. In fact, if a (declassification-free) program is deemed secure by Jif's flow-insensitive analysis, then it is also secure with respect to our flow-sensitive analysis. However, soundness may come at the expense of additional manual annotations due to the flow insensitivity of Jif's analysis.

Another difference with our formal model is the enforcement of delayed declassification. We not do implement the propagation of the declassify flag (d) used to track delayed declassification in section IV-C5. Instead, we require that every `declassify` operator is used only as a parameter of a `return` statement. Since any declassifications are performed within the scope of the return statement, it simply disallows the programmer to write code subject to delayed declassification leaks. We implement this check during the static analysis.

We use the SGX-LKL [38] framework to run the JAR corresponding to the enclave partition. SGX-LKL uses Linux Kernel Library [39] to handle system calls from the application within the enclave. In SGX-LKL, the application JAR, a JVM, and the necessary LKL binaries are compiled to a single image file. SGX-LKL loads the image and runs it inside the enclave.

VII. Evaluation

In this section, we present the evaluation of J_E. We implemented case studies to demonstrate how the design of J_E can address the security requirements of distributed applications. The goal of the case studies is to exhibit the features of the language. Additional case studies are provided in the technical report [32].

A. Password Checker

Listing 7 shows a password checker. The class `PasswordChecker` is annotated with the `@Enclave` annotation. As a result, during run time, the `PasswordChecker` class is placed inside the enclave. The field `password` is annotated with the `@Secret` annotation. The method `checkPassword` returns a boolean which is the result of the comparison of hashes of the field `password` and parameter `guess`. The code snippet without any annotations is a valid Java code. The case study demonstrates the HAA attacker scenario. The parameter `guess` 6 to the gateway method `checkPassword` is controlled by the attacker and is considered untrusted.

B. Apache Spark

Spark [40] is a popular processing framework for big data. It is widely used for machine learning and streaming jobs within clusters. We show a J_E implementation of secure medical data processing using the enclave and Spark. The goal is to process encrypted medical records inside the enclave to extract statistics about the records. Listing 8 represents the code running outside the enclave. We create a Spark context object `sc` (Line 5) and store the encrypted records to be processed in a `JavaRDD` (resilient distributed dataset) object `recList` (Line 7) which is a distributed data structure to store the data to be processed. `EncRec` is an encrypted medical record. The `StatUtil` class (Line 12) contains a gateway method `process` (Line 17) that accepts an encrypted record, it decrypts it with the secret key `key`, and returns the corresponding statistical information `StatRecord`. Hence, a medical record is only decrypted inside the enclave, protecting the medical data. The code without any annotations is a valid Java program, and the programmer only needs to add the annotations without any extra modifications to the original program.

C. Updatable Password Checker

We consider a password checker with a provision for updating the password via a gateway method. Listing 9 shows the class `UpdatablePasswordChecker` annotated with the `@Enclave` annotation. It contains a secret field `password` and two gateway methods, namely, `checkPassword` and `updatePassword`. The `checkPassword` method (Line 12) compares the argument `guess` with the field `password` and returns the result of the comparison as a boolean. Method `updatePassword` (Line 7) assigns the value of the argument `newPass` to the secret field `password` (Line 9). This case study illustrates the HRAA attack scenario. The attacker can reorder the calling sequence of gateway methods. Since the gateway method `updatePassword` assigns an untrusted value to the secret field `password` (Line 9), the secret field can no longer be trusted. Therefore, we infer the security level of the `password` secret field as <$\mathbb{S}$,$\mathbb{U}$>. The field `password` needs to be endorsed (Line 15) to allow the declassification (Line 18). J_E detects secret fields that can be modified by an attacker and the programmer needs to explicitly trust such fields to allow any dependent declassification.

VIII. Related Work

This section compares J_E with closely related work on securing distributed applications with TEEs, focusing on information flow control, frameworks for TEEs, and secure program partitioning.

1) Information flow control and enclaves: Recent works leverage IFC and enclaves to build applications with strong security guarantees. These works target specific challenges in the domain space, including security foundations of enclave applications, secure code partitioning for enclaves and language support for programming in TEEs. J_E pushes the boundary in several directions providing: (a) language support for Java applications in Intel SGX; (b) a security type

Listing 8: Medical data processing using Spark

```
class MainClass {

   public static void main(String[] args) {
      JavaSparkContext sc = new JavaSparkContext(
            new SparkConf().setAppName("Foo"));
      JavaRDD<EncRec> recList =
            sc.parallelize(getEncryptedRecords());
      List<StatRecord> recList = recList.map(x ->
            StatUtil.process(x));
} }
```

```
@Enclave
class StatUtil  {

   @Secret static String key;

   @Gateway
   public static StatRecord process(EncRec rec) {
      EncRec recE = endorse(rec);
      Record rec = decrypt(key, recE);
      // process the decrypted record
      return declassify(rec);
} }
```

Listing 9: Updatable password checker

```
@Enclave
class UpdatablePasswordChecker {

   @Secret static String password;

   @Gateway
   public static void updatePassword(String currPass, String newPass) {
      if (checkPassword(currPass) == true) {
         password = newPass;
   } }
   @Gateway
   public static boolean checkPassword(String guess) {
      boolean result = false;
      String guessE = endorse(guess);
      if (endorse(password).equals(guessE)) {   // endorsing password
         result = true;  // result of type <S,T>
      }
      return declassify(result);  // declassifying result
} }
```

system enabling verification of secure partitioning against realistic attacker models; (c) formalization and soundness proofs with respect to semantic characterization of security; (d) a prototype implementation supported by case studies.

Sinha et al. [15] present Moat, a system for statically verifying confidentiality properties of Intel SGX programs in the presence of passive and active attackers. They combine a flow sensitive type system with automated theorem proving to check confidentiality policies with declassification. Their active attacker is similar to our HAA attacker, assuming arbitrary modification of non-SGX code. By contrast, we focus on verifying robustness of an application against active attacker which has the advantage of using a more lightweight analysis whenever a program is already proved secure against passive attackers. Moreover, we consider more powerful attackers, and target languages with managed runtimes, thus shielding developers from SGX-specific details. Follow-up work by Sinha et al. [41] enforces Information Release Confinement, an access control policy which allows SGX code to perform arbitrary computations ensuring that it can only generate output to the non-SGX memory through encrypted channels. While this approach cannot enforce expressive information flow policies, it implements Control Flow Integrity to ensure that an active attacker does not compromise the control flow of an executions. Our approach would require similar techniques to enforce security against the HRAA attacker.

Inspired by Moat, Gollamudi and Chong [16] propose a security type system to enforce flexible information flow policies for an SGX-enabled imperative language. They abstract away the details of enclave management and develop a translation tool to automatically infer the parts of the program to be executed inside the enclaves. They consider active attackers and formalize security using knowledge-based conditions. By contrast, our approach enforces robustness for a mainstream language like Java. Recent work by Gollamudi et al. [42] uses enclaves to enforce more expressive confidentiality and integrity against passive attackers in a distributed setting. Like us, both works come with soundness proofs of security, with the key difference that our work targets robustness instead of variants of noninterference. Liu et al. [43], [44] study automated program partitioning with SGX for passive attackers. We argue that passive attackers are too weak in the context of SGX, hence our work provides the formal grounds for extending and proving their techniques in a realistic setting.

A large array of works study IFC in distributed settings [19], [45], [46], [47], [48], [49]. These works address the problem of secure program partitioning across nodes in a distributed system under the assumption that low integrity nodes may be controlled by an active attacker. Our work draws inspiration from these approaches and extends them to capture attacker models arising in TEEs.

Existing works target the foundations of IFC in presence of active attackers [50], [21], [51], [52], [53], [20] and enforce robustness via security type systems.

2) Programming frameworks for TEEs: To reduce the trusted computing base, in contrast to running unmodified applications inside SGX, some approaches focus on partitioning the application into components that execute within the enclave and the rest, which executes in the untrusted environment. Glamdring [54] propose the first source-level partitioning framework for securing C applications with Intel SGX. Developers annotate sensitive data and Glamdring automatically partitions the application into untrusted and enclave parts. Panoply [55] generates application binaries from the annotated source code where annotations specify the parts of the application to be run inside separate enclaves. DynSGX [56] provides tools to dynamically load, unload and execute compiled functions inside SGX enclaves efficiently. While these works leverage enclaves to enforce strong isolation properties, they do not enforce application-level policies and lack formal security guarantees.

Several works aim to ease the programming of applications that (partially) execute within an enclave. Coppolino et al. [57] present a comparative analysis of the existing approaches for securing Java applications with Intel SGX. RUST-SGX [58] provides memory-safe SGX support for Rust through a

memory management scheme to control the interface between Rust and Intel's C/C++ APIs. Civet [17] is a programming framework for Java using an XML file to specify the classes that are executed inside the enclave. Civet leverages dynamic information flow control to track insecure flows within the enclave. Uranus [59] supports executing Java functions inside SGX enclaves. It provides two method-level annotations `JECall` and `JOCall` to indicate methods to be executed inside and outside the enclave respectively. Secure Routines [60] is a programming framework for SGX in the Go language. Programmers can execute Go functions (goroutine) inside an enclave, and use low-overhead channels to communicate with the untrusted environments. Like us, these works aim at developing programming models with enclaves. However they lack provable security guarantees and require security analysis of the partitioned application from scratch.

Other works propose processor model calculi to capture necessary conditions for safe remote execution of enclave programs. Subramanyan et al. [61] introduce an abstract processor model to verify the security guarantees of Intel SGX under specific adversary capabilities. Sinha [62] studies confidentiality risks that can be exploited by application and infrastructure attacks in SGX applications. Autarky [63] is a controlled-channel attack resistant framework based on a modified SGX ISA.

3) Running unmodified applications inside SGX: Haven [24] was the first system built to run unmodified Windows legacy applications inside Intel SGX. SCONE [25] is a Docker extension that uses SGX to protect container processes. SGX-LKL [28], SGXKernel [26], Graphene-SGX [27], and Occlum [64] are library OS based frameworks designed to run unmodified Linux applications inside the SGX enclave.

4) Multitier programming and secure program partitioning: The J_E programming model is inspired by the multitier programming paradigm [65], [66], [67] – for a comprehensive overview of multitier programming, we refer to the survey by Weisenburger et al. [68]. In multitier programming, the code for different tiers is written as a single compilation unit and the compiler automatically splits it into the components associated to each individual tier. Different works extend the multitier programming model with information flow policies to build secure web applications via security type systems [69], [70], [71], [72] and symbolic execution [73]. In contrast to J_E, none of these approaches enforce robustness properties.

IX. Conclusion

In this paper we present J_E, a language for confidential computing which supports enclave-enabled applications. First, J_E seamlessly integrates with a high-level, managed language, and enables programmers to develop secure enclave-enabled applications by adding annotations to Java programs. Second, J_E comes with a security model that accounts for realistic attackers, that, in the case of enclave programming, can tamper with the code and the data of the non-enclave environment. We define the notion of *robustness* of enclave-enabled programs and prove that it is correctly enforced by the J_E type system.

We evaluate our approach on several use cases from the literature, including a battleship game, a secure event processing system, and a popular processing framework for big data, showing that it can correctly handle complex cases of partitioning, information flow, declassification and trust.

We envision different avenues for future work including a generalization of our framework to multiple enclaves and nonmalleable information flow. On the practical side, we plan to extend our automated partitioning and compilation algorithms to handle Java programs beyond the core J_E fragment.

Acknowledgments

Thanks are due to Owen Arden and anonymous reviewers for their helpful feedback on this paper. This work is partially supported by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the BRF Project 1025524 from the University of St.Gallen, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

References

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," ser. HASP '13, 2013.
[2] Intel, "Intel® software guard extensions developer guide," 2014, accessed 2021-05-20. [Online]. Available: https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-developer-guide.pdf
[3] ARM, "Building a secure system using trustzone® technology," 2009.
[4] S. Pinto and C. Garlati, "Multi zone security for arm cortex-m devices," 2020, https://hex-five.com/wp-content/uploads/2020/02/Multi-Zone-Security-White-Paper-20200224.pdf.
[5] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," 2016, pp. 857–874.
[6] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016.
[7] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *MICRO'52*, 2019, p. 42–56.
[8] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys'20*, 2020.
[9] Apple, "Apple platform security," https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, 2020, accessed 2021-05-20.
[10] Fortanix, "The fortanix runtime encryption," https://fortanix.com/products/runtime-encryption, accessed 2021-05-20.
[11] Anjuna, "Anjuna enterprise enclaves," https://www.anjuna.io/enterprise-enclaves, accessed 2021-05-20.
[12] Oracle, "Java native interface," https://docs.oracle.com/javase/8/docs/technotes/guides/jni, accessed 2021-05-20.
[13] GraalVM, "Graalvm native image," https://www.graalvm.org/docs/reference-manual/native-image, accessed 2021-05-20.
[14] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 2003.
[15] R. Sinha, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *CCS'15*, 2015, pp. 1169–1184.
[16] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *OOPSLA'16*, 2016, p. 494–513.
[17] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An efficient java partitioning framework for hardware enclaves," in *USENIX Security'20*, 2020.
[18] A. C. Myers, "JFlow: Practical Mostly-static Information Flow Control," in *POPL'99*, 1999, pp. 228–241.
[19] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *SOSP'01*, 2001, p. 1–14.

[20] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification," in *CSFW'04*, 2004, pp. 172–186.

[21] A. Askarov and A. C. Myers, "Attacker control and impact for confidentiality and integrity," *LMCS'11*, 2011.

[22] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL'06*, 2006, pp. 79–90.

[23] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[24] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, p. 8, 2015.

[25] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in *OSDI'16*, 2016, pp. 689–703.

[26] H. Tian, Y. Zhang, C. Xing, and S. Yan, "Sgxkernel: A library operating system optimized for intel sgx," in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 35–44.

[27] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *USENIX ATC'17*, 2017, pp. 645–658.

[28] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," 2019.

[29] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS*, 2008.

[30] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security*, 2016.

[31] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *ACM CCS*, 2010.

[32] A. Oak, A. M. Ahmadian, M. Balliu, and G. Salvaneschi, "Language Support for Secure Software Development with Enclaves," 2021.

[33] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, 2009.

[34] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *S&P'07*, 2007, pp. 207–221.

[35] M. Balliu, I. Bastys, and A. Sabelfeld, "Securing IoT Apps," *IEEE Security & Privacy Magazine*, 2019.

[36] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *CCS'17*, 2017, pp. 1875–1891.

[37] Oracle, "Java remote method invocation - distributed computing for java," https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html, accessed 2021-05-20.

[38] SGX-LKL, "Sgx-lkl library os for running linux applications inside of intel sgx enclaves," https://github.com/lsds/sgx-lkl, accessed 2021-05-20.

[39] O. Purdila, L. A. Grijincu, and N. Tapus, "Lkl: The linux kernel library," in *9th RoEduNet IEEE International Conference*, 2010, pp. 328–333.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Hot Topics in Cloud Computing*, 2010, p. 10.

[41] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *PLDI'16*, 2016, pp. 665–681.

[42] A. Gollamudi, S. Chong, and O. Arden, "Information flow control for distributed trusted execution environments," in *CSF'19*, 2019, pp. 304–30 414.

[43] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *CCS'17*, 2017, pp. 2359–2371.

[44] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," in *CCS'19*, 2019, pp. 1023–1040.

[45] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *SOSP'07*, 2007, pp. 31–44.

[46] J. Liu, O. Arden, M. D. George, and A. C. Myers, "Fabric: Building open distributed systems securely by construction," *Journal of Computer Security*, pp. 367–426, 2017.

[47] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic, "Using replication and partitioning to build secure distributed systems," in *S&P'03*, 2003, p. 236.

[48] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. Comput. Syst.*, p. 283–328, Aug. 2002.

[49] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *USENIX Security 07*. Boston, MA: USENIX Association, Aug. 2007.

[50] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *CCS'17*, 2017, pp. 1875–1891.

[51] M. Balliu and I. Mastroeni, "A Weakest Precondition Approach to Active Attacks Analysis," ser. PLAS '09, 2009, pp. 59–71.

[52] C. Fournet, G. L. Guernic, and T. Rezk, "A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms," in *CCS'09*, 2009, pp. 432–441.

[53] M. Balliu and I. Mastroeni, "A Weakest Precondition Approach to Robustness," *Transactions on Computational Science X*, vol. 10, pp. 261–297, 2010.

[54] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for intel sgx," in *USENIX ATC'17*, 2017, p. 285–298.

[55] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with SGX enclaves," in *NDSS'17*, 2017.

[56] R. Silva, P. Barbosa, and A. Brito, "Dynsgx: A privacy preserving toolset for dinamically loading functions into intel(r) sgx enclaves," in *CloudCom'17*, 2017, pp. 314–321.

[57] L. Coppolino, S. D'Antonio, G. Mazzeo, and L. Romano, "A comparative analysis of emerging approaches for securing java software with intel sgx," *Future Generation Computer Systems*, pp. 620 – 633, 2019.

[58] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with rust-sgx," in *CCS'19*, 2019, p. 2333–2350.

[59] J. Jiang, X. Chen, T. Li, C. Wang, T. Shen, S. Zhao, H. Cui, C. Wang, and F. Zhang, "Uranus: Simple, Efficient SGX Programming and its Applications," in *ASIA CCS'20*, 2020.

[60] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured routines: Language-based construction of trusted execution environments," in *USENIX ATC'19*, 2019, p. 571–585.

[61] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *CCS'17*, 2017, p. 2435–2450.

[62] R. Sinha, "Secure computing using certified software and trusted hardware," Ph.D. dissertation, 2017.

[63] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *EuroSys'20*, 2020.

[64] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *ASPLOS'20*, 2020, p. 955–970.

[65] M. Serrano, E. Gallesio, and F. Loitsch, "Hop, a language for programming the web 2.0," ser. OOPSLA Companion '06. ACM, 2006.

[66] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web programming without tiers," ser. FMCO '06. Springer-Verlag, 2006, pp. 266–296.

[67] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with scalaloci," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276499

[68] P. Weisenburger, J. Wirth, and G. Salvaneschi, "A survey of multitier programming," *ACM Comput. Surv.*, Sep. 2020.

[69] B. J. Corcoran, N. Swamy, and M. W. Hicks, "Cross-tier, label-based security enforcement for web applications." in *SIGMOD*, 2009.

[70] D. Schoepe, D. Hedin, and A. Sabelfeld, "SeLINQ: tracking information across application-database boundaries," in *ICFP*, 2014.

[71] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "JSLINQ: building secure applications across tiers," in *CODASPY'16*, 2016, pp. 307–318.

[72] J. Parker, N. Vazou, and M. Hicks, "Lweb: information flow security for multi-tier web applications," *Proc. ACM Program. Lang.*, vol. 3, no. POPL'19, pp. 75:1–75:30, 2019.

[73] A. Chlipala, "Ur/web: A simple model for programming the web," in *POPL '15*. Association for Computing Machinery, 2015, p. 153–165.