Language Support for Secure Software Development with Enclaves

Aditya Oak  Amir M. Ahmadian  Musard Balliu  Guido Salvaneschi
TU Darmstadt  KTH Royal Institute of Technology  KTH Royal Institute of Technology  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 on 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]. 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 @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 V describes the implementation. The evaluation is in Section VI. Section VII 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) [11], [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

https://prg-grp.github.io/je-lang
We also ignore covert channels including termination and can only observe the entire flow policies and are verified via a security type system. The benefit of this approach is that the developer can keep the partitioned program will be as secure as 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.

### 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

### Listing 1: Language constructs

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

### Listing 2: The Endorse operator

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

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 \( \bar{f} \) for a finite list of elements \( l_1, \ldots, l_n \). 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 \( \bar{f} \) and methods \( \bar{m} \). 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 \( \bar{p} \) 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 \( e.m(\bar{p}) \). Commands \( S \) include standard features such as assignment to class fields and variables, conditionals, loops, and sequencing.
Command \( x := \text{declassify}(e) \) 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 \( x := C.m(p) \) denotes a call to method \( m \) of class \( C \) with actual parameters \( p \), 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 \( \langle S, M, \mathcal{H} \rangle \) consists of a command \( S \), a memory \( M = \text{Vars} \rightarrow \text{Val} \) mapping variables to values, and a heap \( \mathcal{H} = \text{Class} \times \text{Field} \rightarrow \text{Val} \) mapping class and field identifiers to values. A state \( \sigma = \langle M, \mathcal{H} \rangle \) is a pair of a memory \( M \) and a heap \( \mathcal{H} \). An observation trace \( t \) is a (possibly empty) sequence of events \( \beta \in \text{Val} \), and \( t_1 \cdot t_2 \) denotes trace concatenation.

We use judgments of the form \( \alpha \vdash_\delta \langle S, M, \mathcal{H} \rangle \Downarrow \langle M', \mathcal{H}' \rangle \) to denote that a configuration \( \langle S, M, \mathcal{H} \rangle \) evaluates to memory \( M' \) and heap \( \mathcal{H}' \) in execution mode \( \alpha \in \{N, E\} \) with static mode \( \delta = \text{Class} \cup \text{Var} \rightarrow \{N, E\} \), and produces an observation trace \( t \). We write \( \alpha \vdash_\delta \langle S, M, \mathcal{H} \rangle \Downarrow \mathcal{H}' \) if the observation trace is empty and \( \alpha \vdash_\delta \langle S, M, \mathcal{H} \rangle \Downarrow \_ \) to ignore the final state. We use modes to distinguish between the enclave environment (\( E \)) and the non-enclave environment (\( N \)). An execution mode \( \alpha \) indicates the current execution environment of a configuration, while a static mode \( \delta \) associates a class identifier or a variable with the execution environment it was assigned to, statically. Abusing notation, we write \( \alpha \vdash_\delta \langle S; \text{return}(e), \mathcal{M}, \mathcal{H} \rangle \Downarrow \mathcal{M}, \mathcal{H} \forall v \) for commands that yield a value \( v \), via the \text{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 \( \langle M_0, \mathcal{H}_0 \rangle \), i.e., \( N \vdash_\delta \langle S_0, M_0, \mathcal{H}_0 \rangle \).

The semantics of expressions, Figure 2 is mostly standard. We use judgments of the form \( \alpha \vdash \langle e_G, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle v, M', \mathcal{H}' \rangle \) to denote that an expression \( e_G \) evaluates to value \( v \), memory \( M' \), and heap \( \mathcal{H}' \) in state \( \langle M, \mathcal{H} \rangle \), execution mode \( \alpha \), static mode \( \delta \), and it emits the event \( \beta \). Rule \text{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 \text{getMethod} : \text{Class} \times \text{Method} \rightarrow m \) to extract a method definition, and \( \text{fields} : \text{Class} \rightarrow \psi(\text{Field}) \) 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, \( \mathcal{H}[\langle C, f \rangle \mapsto v] \) 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 [2].

Rule \text{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 \text{METHOD} evaluates expression \( e \) to a class identifier \( C \) and uses the \text{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 (\( \alpha = N \wedge \delta(C) = E \wedge \delta = G \)). 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 \( \epsilon \).

Figure 3 provides an excerpt of evaluation rules for \( J_E \) commands. The full set of rules can be found in the technical report [2]. Rule \text{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 \text{STORE} emits is equal to the event emitted during the evaluation of expression \( e_G \). Rule \text{RETURN} evaluates the expression in the current state and returns the associated value.

### C. Security Model

1) \text{Security Policy:} \( J_E \) adopts security labels to specify application-level policies as information flow policies. A security label \( \ell \) is a tuple \( (\ell_C, \ell_I) \) of a confidentiality label \( \ell_C \) and an integrity label \( \ell_I \). We use two levels, \text{Public} (\( P \)) and \text{Secret} (\( S \)), for confidentiality, and two levels, \text{Trusted} (\( T \)) and \text{Untrusted} (\( U \)), for integrity. Intuitively, for confidentiality, data from \text{Secret} sources should not flow to \text{Public} sinks unless it is explicitly declassified by the developer, and, for integrity, data from \text{Untrusted} sources should not flow to \text{Trusted} sinks unless it is explicitly endorsed by the developer. These requirements are reflected by the ordering relation between security labels, namely \( P \subseteq S \) and \( T \subseteq U \). The product lattice \( L \) lifts the constraints to an ordering relation over label pairs such that \( \ell_1 \subseteq \ell_2 \) if \( \ell_1.C \subseteq \ell_2.C \) and \( \ell_1.I \subseteq \ell_2.I \). The lattice \( L \) 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) \text{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 \( L \) as defined by a security mapping \( \Gamma : \{\text{Var} \cup \text{Class} \cup \text{Field}\} \rightarrow L \). We then define
indistinguishability over pairs of program states for a security mapping $\Gamma$ and an attacker at security level $A = (P, U)$.

**Definition 1 (State Indistinguishability).** Two memories $M_1$ and $M_2$ are indistinguishable for the attacker $A$ (written $M_1 =_A M_2$) iff $\forall x. \delta(x) = N, M_1(x) = M_2(x)$, and $\forall x. \delta(x) = E$ such that $\Gamma(x) = (P, -), M_1(x) = M_2(x)$.

Two heaps $H_1$ and $H_2$ are indistinguishable for the attacker $A$ (written $H_1 =_A H_2$) iff $\forall C. \delta(C) = N, \forall f \in \text{fields}(C), H_1(C, f) = H_2(C, f)$ and $\forall C. \delta(C) = E, \forall f \in \text{fields}(C)$ such that $\Gamma(C, f) = (P, -), H_1(C, f) = H_2(C, f)$.

Two program states $\sigma_1 = (M_1, H_1)$ and $\sigma_2 = (M_2, H_2)$ are indistinguishable for the attacker $A$ (written $\sigma_1 =_A \sigma_2$) iff $M_1 =_A M_2$ and $H_1 =_A H_2$.

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 $\beta$. 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 $\sigma_1$ and $\sigma_2$ be two initial states such that $\sigma_1 =_A \sigma_2$. Two executions are indistinguishable (written $N \vdash_\beta (S, \sigma_1) \approx_A N \vdash_\beta (S, \sigma_2)$) if $N \vdash_\beta (S, \sigma_1) \Downarrow t_1$ and $N \vdash_\beta (S, \sigma_2) \Downarrow t_2$ 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.

![Figure 2: Excerpt of $J_E$ expression semantics](image1)

![Figure 3: Excerpt of $J_E$ command semantics](image2)
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 $[\bullet]; x := e.m(p)$ and $[\bullet]; C.f := e.m(p)$, and define the active attacks.

**Definition 3 (J_E program with holes).** A program with holes $S'[\bullet]$ is defined by extending the syntax in Figure 7 as follows:

$$S'[\bullet] := \ldots | [\bullet]; x := e.m(p) | [\bullet]; C.f := e.m(p)$$

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:

$$a := \text{skip} | a_1; a_2 | q := e \ (where \ q \in p) \ (1)$$

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 [7] 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 $S'[\text{skip}]$.

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:

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

where $m$ is a gateway method defined in $S'_1[\overrightarrow{\ell}]$.

In this model, the attack definition of [7] will remain unchanged, indicating that HRAA attacker subsumes the power of HAA attacker.

**Lemma 2.** The attacker code defined in [7] 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
Definition 5 (Robustness under HAA). Program \( S[\tau] \) is robust w.r.t an HAA attacker \( A \) if for all \( \sigma_1, \sigma_2, \tau_1, \tau_2 \):

\[
N \vdash <S[\tau_1], \sigma_1> \approx_A N \vdash <S[\tau_1], \sigma_2> \implies N \vdash <S[\tau_2], \sigma_1> \approx_A N \vdash <S[\tau_2], \sigma_2>
\]

Robustness holds whenever for an attack vector \( \tau \) and two indistinguishable initial states \( \sigma_1, \sigma_2 \), if the program \( S[\tau_1] \) satisfies execution indistinguishability (definition 2), then for another attack vector \( \tau_2 \), the program \( S[\tau_2] \) also satisfies execution indistinguishability. In other words, the attacker observations of \( S[\tau_2] \)'s observation traces do not reveal any secrets apart from what the attacker already knows by the observation traces of the program \( S[\tau_1] \). The PA attacker is captured by executions of the program \( S[\text{skip}] \).

We extend robustness to capture HRAA attackers.

Definition 6 (Robustness under HRAA). Program \( S[\tau'] \) is robust w.r.t an HRAA attacker \( A \) if for all \( \sigma_1, \sigma_2, \tau_1, \tau_2 \) and for all \( S'[\tau'] \):

\[
N \vdash <S'[\tau_1], \sigma_1> \approx_A N \vdash <S'[\tau_1], \sigma_2> \implies N \vdash <S'[\tau_2], \sigma_1> \approx_A N \vdash <S'[\tau_2], \sigma_2>
\]

This definition ensures that the HRAA attacker does not learn more information by changing the order and frequency of gateway calls. Observe that if \( S'[\alpha] \) triggers an execution that was not possible \( S[\tau] \), 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 \( S[\tau] := [a_1]; \text{FooClass.bar}(); [a_2]; \text{FooClass.foo}() \) is not robust under HRAA because the program \( S'[\alpha] := [a_1']; \text{FooClass.foo}(); [a_2']; \text{FooClass.bar}() \) enables the HRAA attacker to reveal the declassified value of secret2. A similar argument applies to calling gateway methods that were defined in \( S[\tau'] \) 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 \( a ::= \text{input} ::= 0 \) and \( a' ::= \text{input} ::= 1 \), 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 \( (S,T) \) as they contain sensitive information and are protected by the enclave. The security label of the arguments of a gateway method is defaulted to \( (P,U) \) and while the return parameter is labeled as \( (P,T) \) or \( (P,U) \).

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 \( \Gamma : (\text{Var} \cup \text{Class} \cup \text{Field}) \rightarrow \mathcal{L} \) mapping variables, classes, and fields to security labels from the security lattice \( \mathcal{L} \). We also use another environment \( \Pi : (\text{Var} \cup \text{Class} \cup \text{Field}) \rightarrow \mathcal{B} \) 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 \( (\ell, d) \). We use indexes \( \ell \) and \( d \) to access elements of this tuple. e.g., \( pc_\ell \) will
show the security label of $pc$. Methods are typed as a set of expressions $\ell, d \vdash S \Sigma (\bullet) : \Gamma, \Pi'$ where $\ell$ is a label of the method, $d$ is a label of the return value, $S$ is a set of expressions, and $\Sigma (\bullet)$ is a set of fields.

The rule $T$-S STORE-GATEWAY-CALL handles the type checking of gateway methods call from a store. The security labels of gateway parameters are explicitly defined in $\Gamma^-$ and must have the $(\mathcal{P}, \mathcal{U})$ security label $\forall p \in \mathcal{P}, \Gamma^- (p) = (\mathcal{P}, \mathcal{U})$, and their flag (defined in $\Gamma^-$) should be false $\forall p \in \mathcal{P}, \Pi^-(p) = F$. The rule ensures that the gateway can only return public values, and the labels of actual parameters are less restrictive than the predefined labels of formal parameters $\Gamma(q_i) \subseteq \Gamma^- (p_i)$, and the method's typing environment is satisfied $\forall y \in dom(\Gamma^-), \Gamma(y) \subseteq \Gamma^-(y)$). Additionally, the typing environment after type checking the method body should respect the method's predefined post typing environment $\forall y \in dom(\Gamma^-), \Gamma(y) \subseteq \Gamma_{out}(y)$, 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 $\Pi$ environment. At last, the rule updates the flag of field $\text{C.f}$ 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 1 in the previous section. We prove that our security type system enforces robustness for the HAA attacker.

**Theorem 1.** If $pc, \Gamma, \Pi \vdash S (\bullet) : \Gamma', \Pi'$ then $S (\bullet)$ 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 $\Sigma$ containing all global variables and fields that are assigned to in at least one method and accessed in at least one method. (i.e., $\Sigma$ contains all global variables and fields that are used and modified in one or more gateways methods) We compute $\Sigma$ 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 $\Sigma$ as untrusted. i.e., $\forall x \in \Sigma, \Gamma_0 [x \mapsto (\Gamma_0 (x) \subseteq \mathcal{U})]$ and $\forall C.f \in \Sigma, \Gamma_0 (C.f) \mapsto (\Gamma_0 (C.f) \subseteq \mathcal{U})$. 

The rule $T$-DECLASSIFY ensures that only trusted data is allowed to be declassified $\ell \subseteq (\mathcal{S}, \mathcal{T})$ and declassification can only happen in public and trusted context $pc \subseteq (\mathcal{P}, \mathcal{T})$. This prevents attacker-controlled untrusted data from influencing the decision to declassify secret information. The security label of variable $x$ will be $(\mathcal{P}, \mathcal{T})$ 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 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 $\Gamma^-$ and must have the $(\mathcal{P}, \mathcal{U})$ security label $\forall p \in \mathcal{P}, \Gamma^- (p) = (\mathcal{P}, \mathcal{U})$, and their flag (defined in $\Gamma^-$) should be false $\forall p \in \mathcal{P}, \Pi^-(p) = F$. The rule ensures that the gateway can only return public values, and the labels of actual parameters are less restrictive than the predefined labels of formal parameters $\Gamma(q_i) \subseteq \Gamma^- (p_i)$, and the method's typing environment is satisfied $\forall y \in dom(\Gamma^-), \Gamma(y) \subseteq \Gamma^-(y)$). Additionally, the typing environment after type checking the method body should respect the method's predefined post typing environment $\forall y \in dom(\Gamma^-), \Gamma(y) \subseteq \Gamma_{out}(y)$, 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 $\Pi$ 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 1 in the previous section. We prove that our security type system enforces robustness for the HAA attacker.

**Theorem 1.** If $pc, \Gamma, \Pi \vdash S (\bullet) : \Gamma', \Pi'$ then $S (\bullet)$ 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 $\Sigma$ containing all global variables and fields that are assigned to in at least one method and accessed in at least one method. (i.e., $\Sigma$ contains all global variables and fields that are used and modified in one or more gateways methods) We compute $\Sigma$ 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 $\Sigma$ as untrusted. i.e., $\forall x \in \Sigma, \Gamma_0 [x \mapsto (\Gamma_0 (x) \subseteq \mathcal{U})]$ and $\forall C.f \in \Sigma, \Gamma_0 (C.f) \mapsto (\Gamma_0 (C.f) \subseteq \mathcal{U})$. 

The rule $T$-DECLASSIFY ensures that only trusted data is allowed to be declassified $\ell \subseteq (\mathcal{S}, \mathcal{T})$ and declassification can only happen in public and trusted context $pc \subseteq (\mathcal{P}, \mathcal{T})$. This prevents attacker-controlled untrusted data from influencing the decision to declassify secret information. The security label of variable $x$ will be $(\mathcal{P}, \mathcal{T})$ 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 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.
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 2 is rejected by our type system. After the preprocessing phase, \( \Sigma = \{ \text{releaseTrigger} \} \) since \text{releaseTrigger} is assigned to in one method call and the program in Listing 4 is rejected by our type system.

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 \( S[\overline{\cdot}] \) denoted by \( G^D \) such that their return value’s flag is true (i.e., \( \Gamma, \Pi \vdash_\delta C.m(\overline{\cdot}) : (-)_{\text{\&}\neg T} \)).

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 \( \text{pathsGW}(S[\overline{\cdot}]) \) to denote the set of all possible paths of \( S[\overline{\cdot}] \), and \( \text{pathGW}_i \) for the set of gateway calls in path \( i \).

3) Check that \( G^D \) is a subset of all of the possible paths of \( S[\overline{\cdot}] \). In other words:

\[
G^D \subseteq \text{pathGW}_i \quad \forall i \in \text{pathsGW}(S[\overline{\cdot}])
\]

This process is performed after calculating \( \Sigma \) 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 5 presents a program with both interfering and noninterfering runs. The program satisfies robustness (Definition 3) and is also accepted by the type system. In fact, if variable \text{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 \text{[choice=v;ComputeArray.compute(choice)]}; will observe the result “Done with computation” + choice. Otherwise, if \text{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.
Theorem 2. If $\text{pc}, \Gamma, \Pi \vdash_\delta S[\sigma] : \Gamma', \Pi'$ with regard to $\Sigma$ and $G^D$, then $S[\sigma]$ 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. 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 trigger-action enclaves to gateways methods that do not 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 $x := \text{endorse}_\eta(e)$. Each endorsement command has a unique label $\eta$, and produces an endorsement event $\text{endorse}(\eta, v)$, which records the label $\eta$ along with the endorsed value $v$.

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 $S[\sigma]$, starting state $\sigma$, and attacker vector $\vec{a}$ which produces trace $t$ (i.e., $\langle S[\vec{a}], \sigma \rangle \downarrow t$), relevant attacks, denoted by $\Omega(S[\vec{a}], \sigma)$, 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 $\text{endorse}(\eta_i, v_i)$, define a set of irrelevant traces based on the number of endorsements in $t$

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

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

We define $\psi(t) = \bigcup \psi_i(t)$ 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 $S[\sigma]$, starting state $\sigma$, and attacker vector $\vec{a}$ such that $\langle S[\vec{a}], \sigma \rangle \downarrow t$, relevant attacks $\Omega(S[\vec{a}], \sigma)$ are defined as:

$$\Omega(S[\vec{a}], \sigma) = \{a' \mid (S[a'], \sigma) \downarrow t' \land t' \notin \psi(t)\}$$

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 $S[\sigma]$ is robust w.r.t an HAA attacker $A$ if for all $\sigma_1, \sigma_2, \vec{a}_1$ and for all $\vec{a}_2 \in \Omega(S[\vec{a}_1], \sigma_1)$

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

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
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 $\text{endorse}_e(e)$ can only be used inside the enclave environment, and endorsement can only occur in a public and trusted context $pc \subseteq (P, T)$. 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.

\textbf{Theorem 3.} If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\emptyset}] : \Gamma', \Pi'$ then $S[\overrightarrow{\emptyset}]$ 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 $S'[\overrightarrow{\emptyset}]$ and $S[\overrightarrow{\emptyset}]$.

In the new definition of relevant attacks, we use the unique endorsement label $\eta$ to ensure that the value of an endorsement used in $S'[\overrightarrow{\emptyset}]$ is equal to the value of that same endorsement in $S'[\overrightarrow{\emptyset}]$ 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.

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

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

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.

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

$$\Omega_R(S[\overrightarrow{a}], S'[\overrightarrow{\emptyset}], \sigma) = \{ \alpha' | (S'[\overrightarrow{a}], \sigma) \downarrow_{t'} \land t' \notin \psi(t) \}$$

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

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

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

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.

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

\section{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.

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.

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
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 do not implement the propagation of the declassify flag \( d \) used to track delayed declassification in Section IV–C. 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 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 3) 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 \( s \), 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 PasswordChecker 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 an untrusted value 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 \( <S,U> \). The field password needs to be endorsed (Line 13) 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...
While this approach cannot enforce expressive information
security type system to enforce flexible information flow policies
an active attacker does not compromise the control flow 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

```java
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));
    }
}

Listing 9: Updatable password checker

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
    }
}
```
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-LKLL [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
