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

Trusted Execution Environments (TEEs)

- Recent Privacy Enhancing Technology
- ‘Enclaves’ protected by hardware
- Enclaves are opaque to the OS

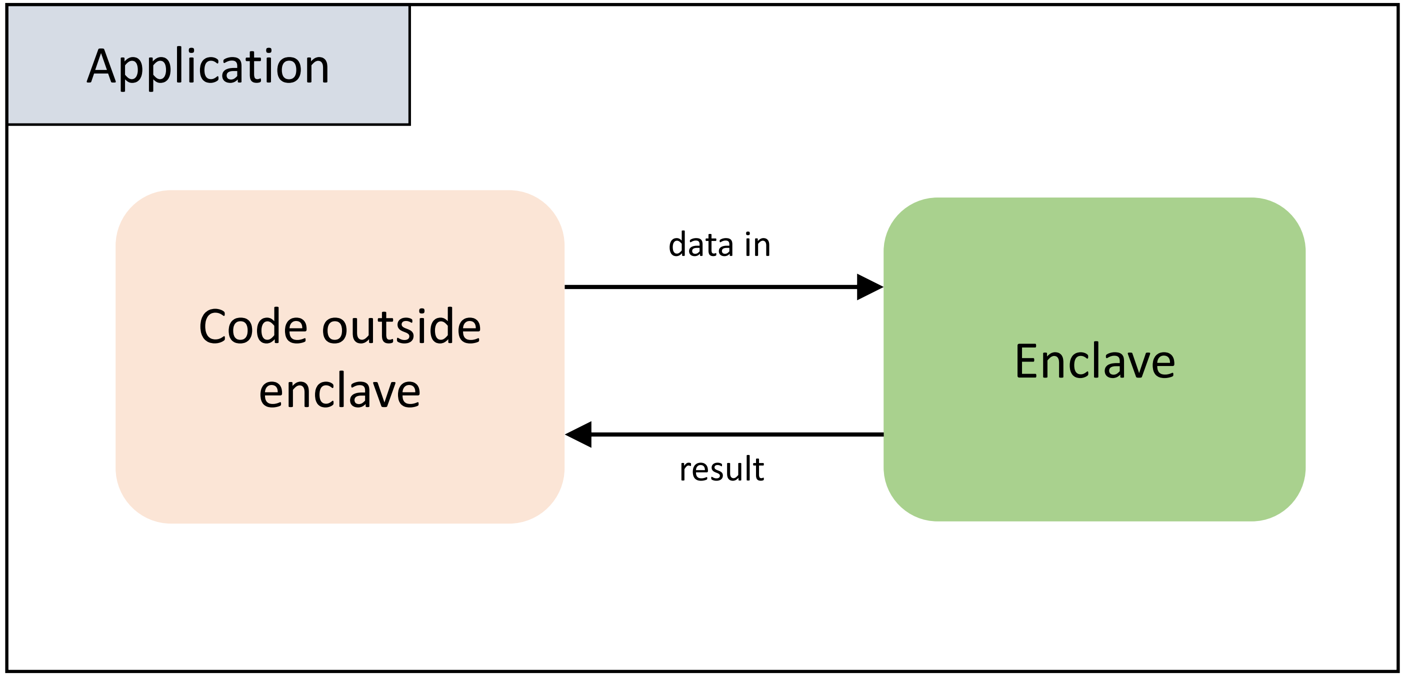
intel SGX

arm TRUSTZONE

Apple Platform Security

- Private computations on cloud hosts

Programming with TEEs



Problems with using TEEs

- Manual application partitioning
- Program the enclave – non-enclave interface
- Application partitioning may not preserve security guarantees

Compiler can do these !!!

J_E: Language support for programming with enclaves

- Language-level abstractions
- Automatic application partitioning
- A static type system for information flow control
- Robustness guarantees against active attackers

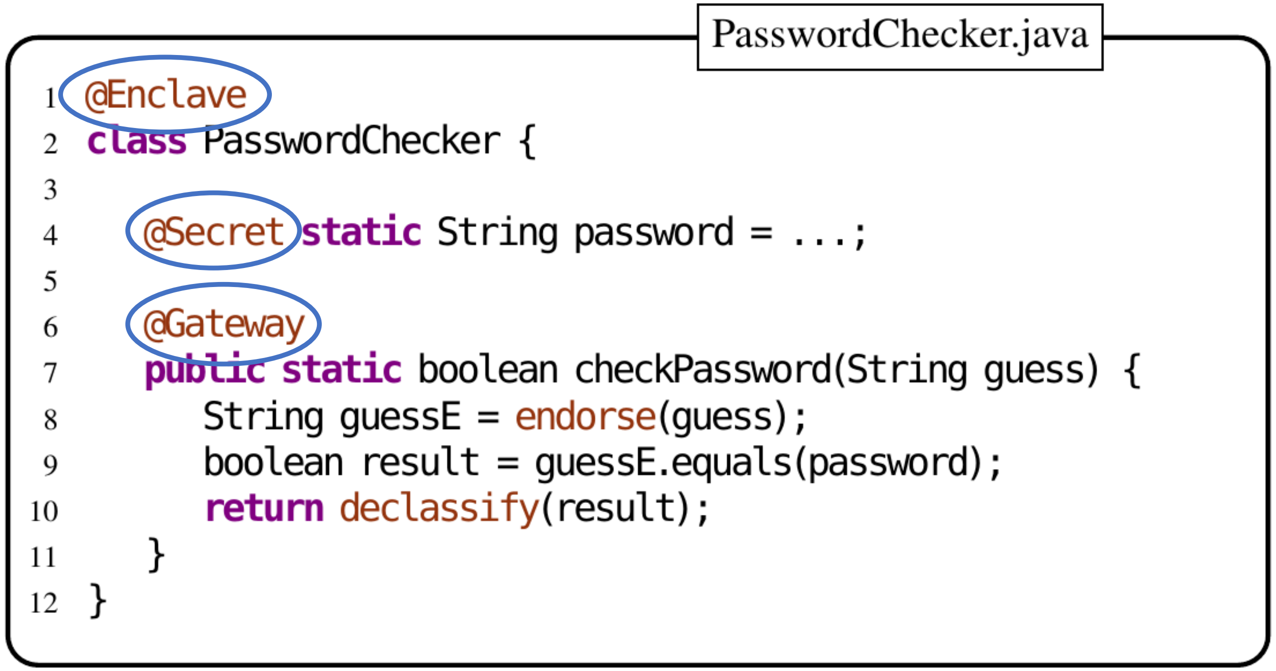
J_E Programming Model: Language-level abstractions

Annotations:

- @Enclave
- @Gateway
- @Secret

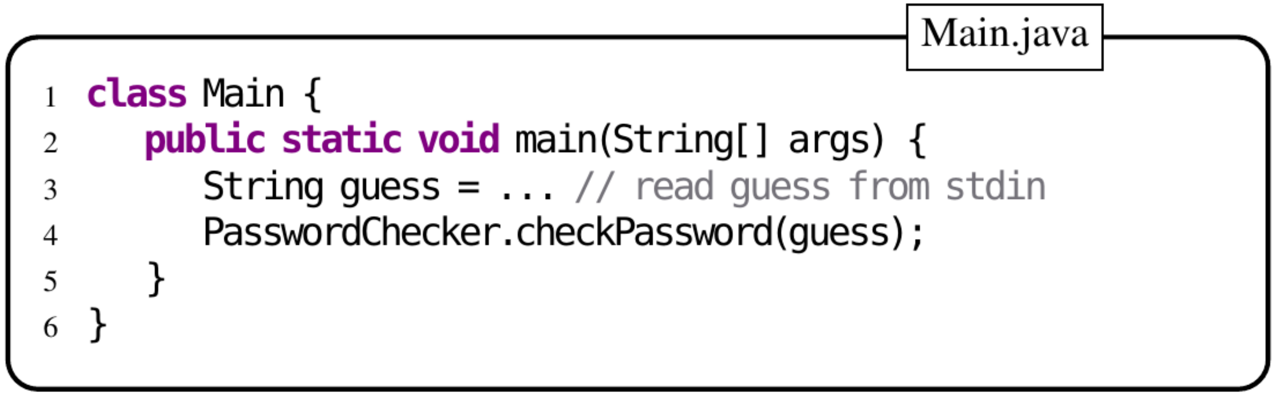
Methods:

- endorse
- declassify



PasswordChecker.java

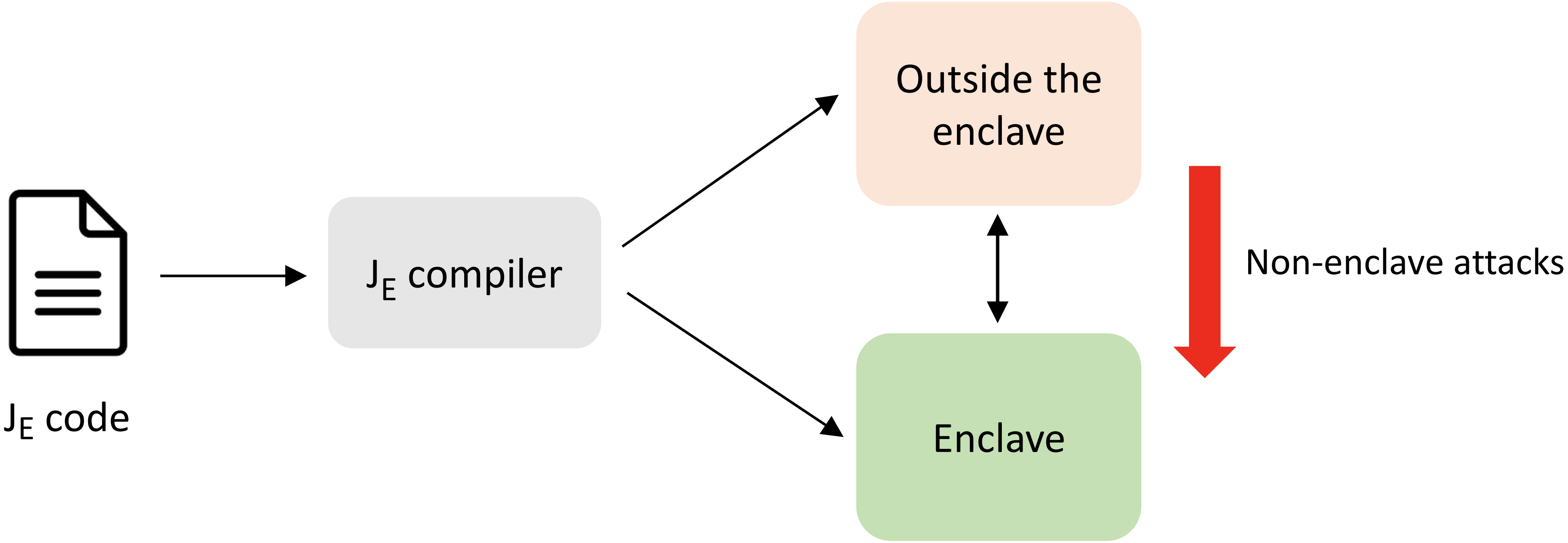
```
@Enclave
class PasswordChecker {

   @Secret static String password = ...;

   @Gateway
   public static boolean checkPassword(String guess) {
      String guessE = endorse(guess);
      boolean result = guessE.equals(password);
      return declassify(result);
   }
}
```



Main.java

```
class Main {
   public static void main(String[] args) {
      String guess = ... // read guess from stdin
      PasswordChecker.checkPassword(guess);
   }
}
```

J_E: Program partitioning and attacker Model



J_E code

J_E compiler

Outside the enclave

Enclave

Non-enclave attacks

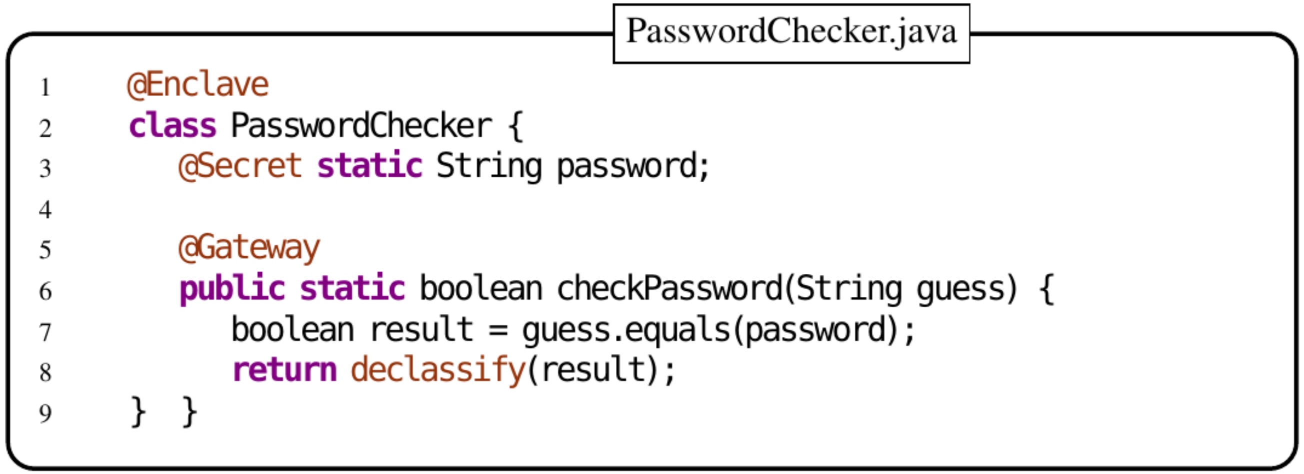
J_E: Attacker models

- Stronger Attackers
  - They can modify data- and code-memory
- Robustness
  - Controlling the release of secret data (declassification)

1. Data-memory attacker (HAA)
2. Data- and code-memory attacker (HRAA)

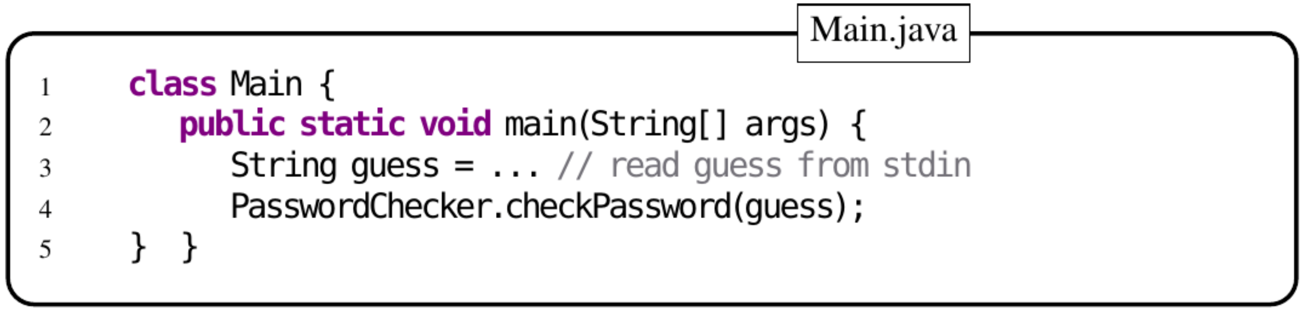
J_E: Attacker models

- HAA Attacker
  - In control of data-memory
  - Change data outside of enclave
    - Parameters of gateways



PasswordChecker.java

```
@Enclave
class PasswordChecker {
   @Secret static String password;

   @Gateway
   public static boolean checkPassword(String guess) {
      boolean result = guess.equals(password);
      return declassify(result);
} }
```



Main.java

```
class Main {
   public static void main(String[] args) {
      String guess = ... // read guess from stdin
      PasswordChecker.checkPassword(guess);
} }
```

J_E: Attacker models

Robustness under HAA

program $\mathrm{S}[\vec{\bullet}]$ *is robust w.r.t* ***HAA*** *attacker* A *if for all* $\sigma_1, \sigma_2, \vec{a}_1, \vec{a}_2$:

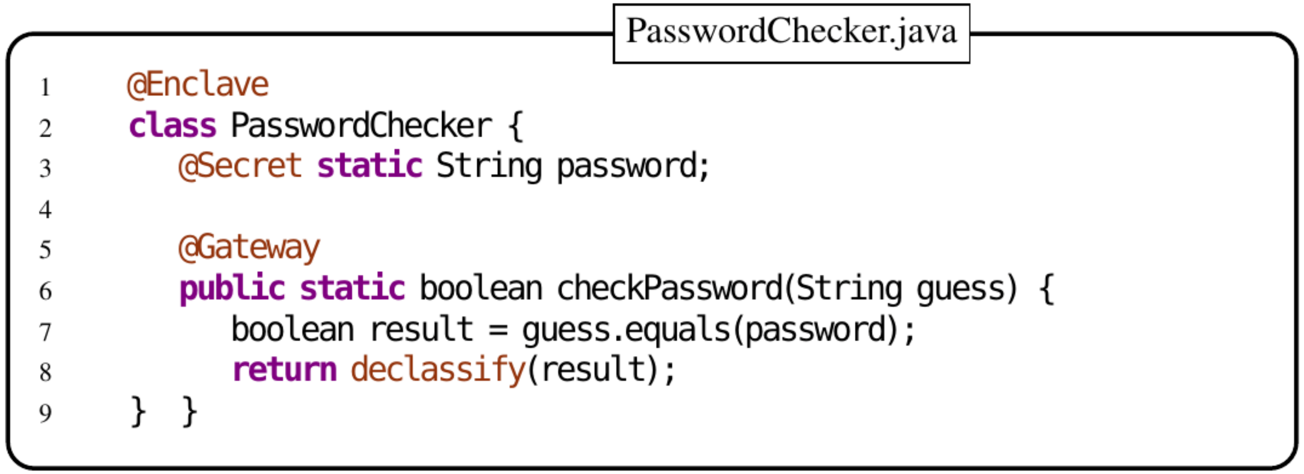
$$N \vdash_\delta \langle \mathrm{S}[\vec{a}_1], \sigma_1\rangle \ \approx_A N \vdash_\delta \langle \mathrm{S}[\vec{a}_1], \sigma_2\rangle \ \Rightarrow$$

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

- We use holes $\vec{\bullet}$ outside of the enclave to capture the effects of this attacker
- $S[\vec{\bullet}]$ is the program
- Attacker’s code a will be placed in these holes
    - It can only affect data-memory

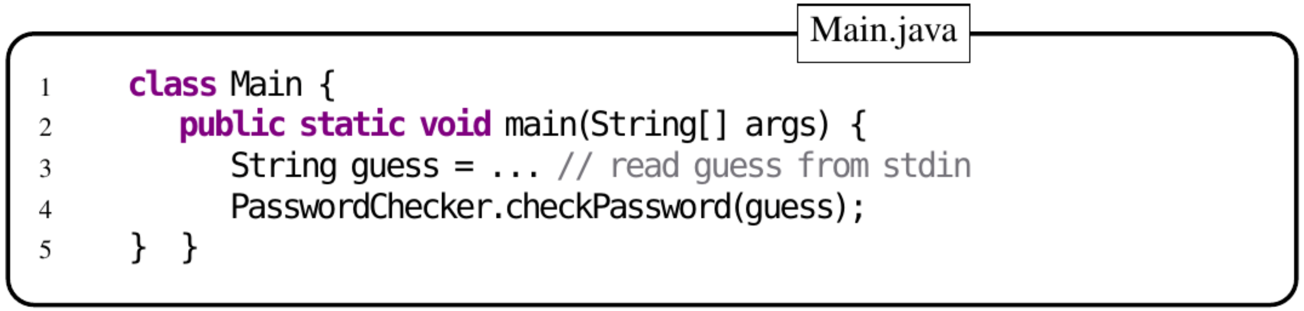
J_E: Attacker models

- Robustness under HAA
  - Parameters of gateways are under attacker’s control
  - They are untrusted
  - Untrusted values should not affect declassification



PasswordChecker.java

```
@Enclave
class PasswordChecker {
   @Secret static String password;

   @Gateway
   public static boolean checkPassword(String guess) {
      boolean result = guess.equals(password);
      return declassify(result);
} }
```



Main.java

```
class Main {
   public static void main(String[] args) {
      String guess = ... // read guess from stdin
      PasswordChecker.checkPassword(guess);
} }
```

J_E: Attacker models

- Enforcing Robustness under HAA
  - Type system ensures that:
    - Only trusted values can be declassified
    - Declassification can only happen under trusted context

$$\textsc{T-Declassify}\ \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \boxed{\ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle} \qquad \boxed{pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle} \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := declassify(e) : \Gamma\left[x \mapsto \ell \sqcap \langle \mathbb{P}, \mathbb{T} \rangle\right], \Pi\left[x \mapsto T\right]}$$

J_E: Attacker models

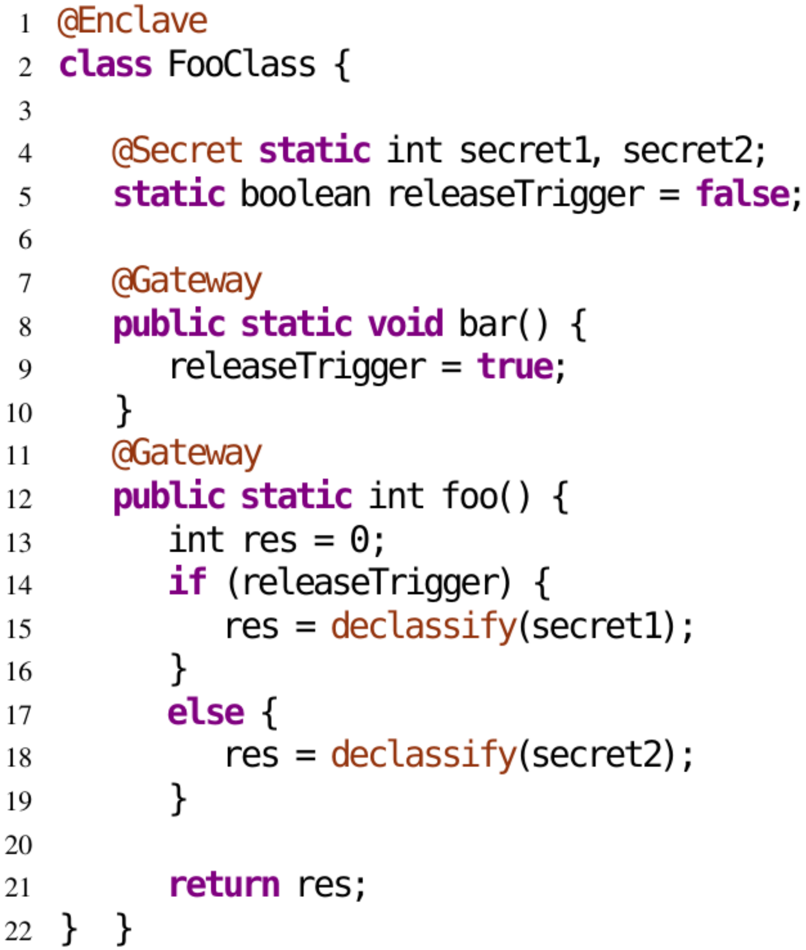
Theorem 1

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

- A well-typed program $pc, \Gamma, \Pi \vdash_\delta S[\vec{\bullet}]: \Gamma', \Pi'$ is robust against HAA attacker.

J_E: Attacker models

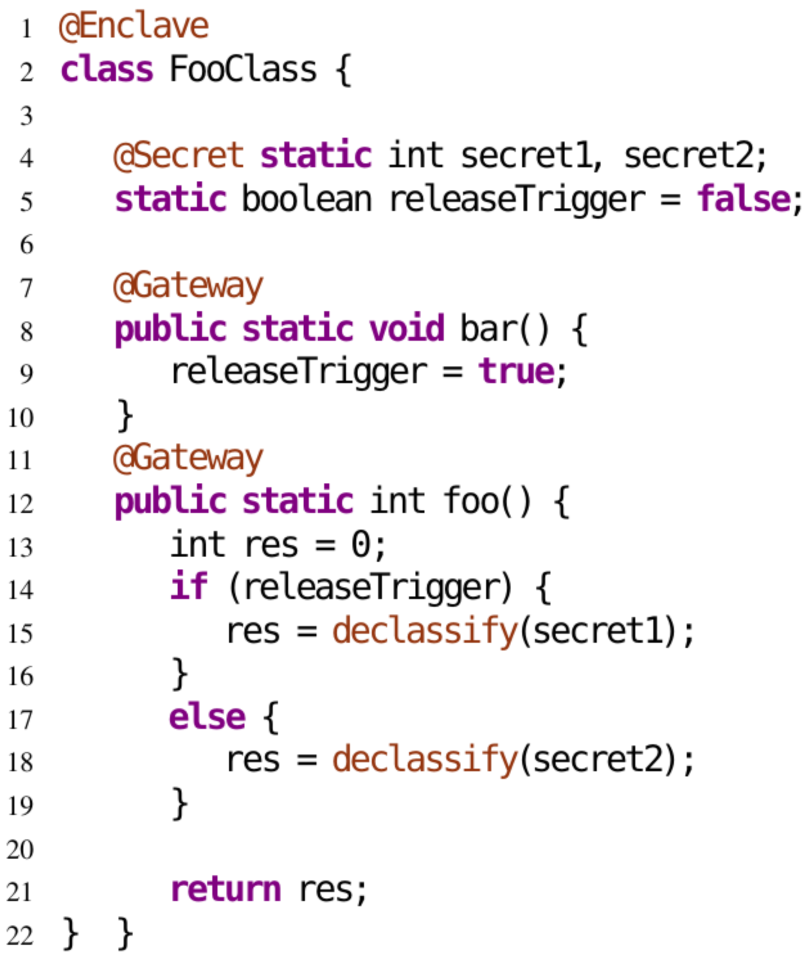
- HRAA Attacker
  - In control of **data**-memory and **code**-memory
  - Can change data and control flow **outside** of enclave
    - Parameters of gateways
    - Order of calling gateways
    - Frequency of calling gateways



```
@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) {
         res = declassify(secret1);
      }
      else {
         res = declassify(secret2);
      }

      return res;
} }
```

J_E: Attacker models

- Robustness under HRAA
  - `foo;bar` or `bar;foo`
    - Can lead to different values
    - Attacker learns more by changing the order of gateway calls



```
@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) {
         res = declassify(secret1);
      }
      else {
         res = declassify(secret2);
      }

      return res;
} }
```

J_E: Attacker models

Program Under HRAA

We define the program under HRAA *attacker as a sequence of gateway calls:*

$$S'[\vec{\bullet}] ::= S_1'[\vec{\bullet}]; S_1'[\vec{\bullet}] \mid [\bullet]; x \coloneqq C.m(\bar{p})$$

- $S'[\vec{\bullet}]$ models attacker's control over **code-memory**
- Holes $\vec{\bullet}$ and attacker's code a model attacker's control over **data-memory**

J_E: Attacker models

Robustness under HRAA

Program $\mathrm{S}[\vec{\bullet}]$ *is robust w.r.t.* ***HRAA*** *attacker* A *if for all* $\sigma_1, \sigma_2, \vec{a}_1, \vec{a}_2$ *and for all* $\mathrm{S}'[\vec{\bullet}]$*:*

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

- $S'[\vec{\bullet}]$ is the attacker program
  - A list of gateway calls

J_E: Attacker models

- We extend the type system to account for this attacker.

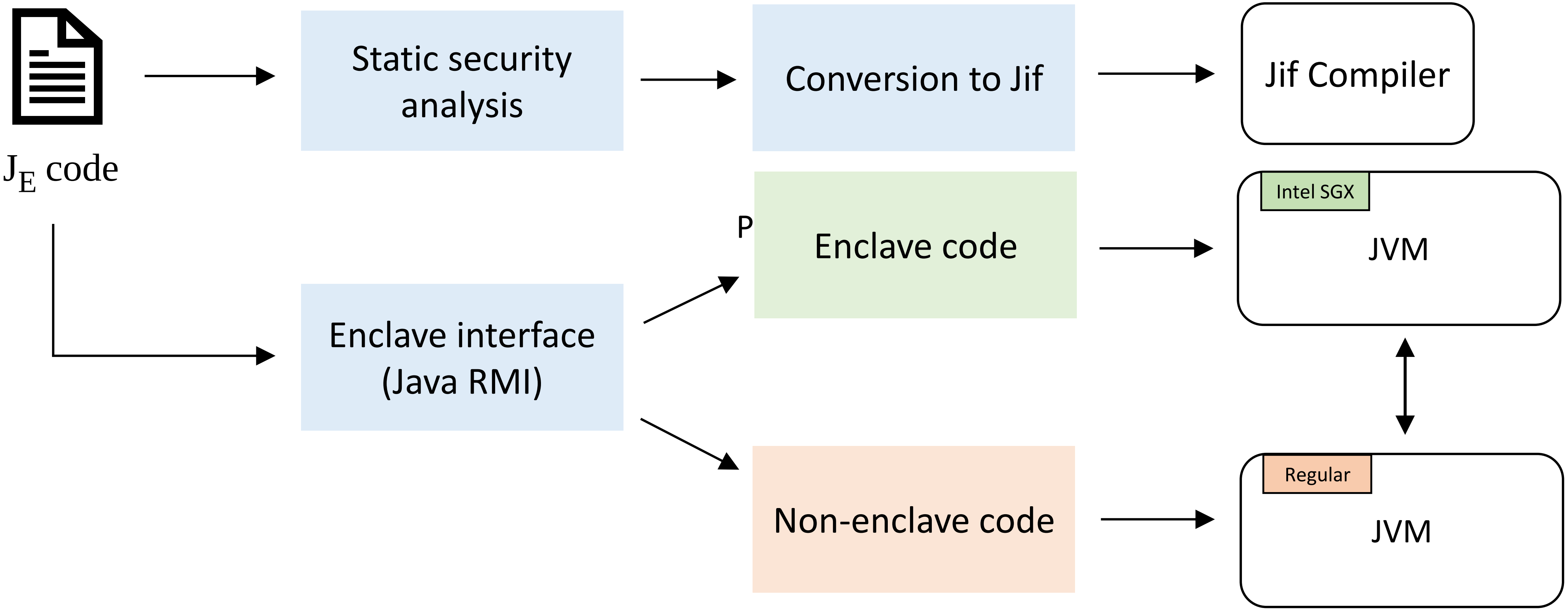
Theorem 2

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

More Investigated Features

- Endorsement
- Flow Sensitive Variables
- Delayed Declassification

J_E Implementation and Workflow



J_E code

Static security analysis

Conversion to Jif

Jif Compiler

Enclave interface (Java RMI)

P

Enclave code

Intel SGX

JVM

Non-enclave code

Regular

JVM

Phase 2

Evaluation

- **Password Checker**: Password stored inside the enclave
- **Updatable-Password Checker**: Password inside the enclave, modifiable from outside
- **Medical Data Processing**: Decrypt and process data inside the enclave

J_E

- A programming model for **secure programming with enclaves**
- Abstractions for **code placement** and **data security attributes**
- Type system to defend against **strong realistic** attacks

More in the paper

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

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

https://prg-grp.github.io/je-lang