Enclave-Based Secure Programming with J_E

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*—Over the past few years, major hardware vendors have started offering processors that support Trusted Execution Environments (TEEs) allowing confidential computations over sensitive data on untrusted hosts. Unfortunately, developing applications that use TEEs remains challenging. Current solutions require using low-level languages (e.g., C/C++) to handle the TEE management process manually – a complex and error-prone task. Worse, the separation of the application into components that run inside and outside the TEE may lead to information leaks. In summary, TEEs are a powerful means to design secure applications, but there is still a long way to building secure software with TEEs alone.**

In this work, we present J_E, a programming model for developing TEE-enabled applications where developers only need to annotate Java programs to define application-level security policies and run them securely inside enclaves.

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

I. Introduction

In cloud computing, cloud service providers offer their infrastructure as a service, and clients use it on an ad-hoc basis. This approach ensures on-demand computing and storage provisioning, but it comes at the price of trusting the cloud providers with potentially sensitive data. Nevertheless, the cloud computing paradigm entails many security and privacy concerns as data is inevitably processed on third-party machines. For example, the cloud could be compromised, but also, the cloud infrastructure may not have strict access control policies to rule out unauthorized access of data. Traditional privacy-preserving techniques struggle to mitigate such issues. For example, symmetric and asymmetric cryptography require encrypted data to be first decrypted to perform any computations – making plaintext data accessible to the hosting infrastructure. On the other hand, homomorphic encryption schemes [1] allow performing computations directly on the encrypted data, but their high computation time and large ciphertext size can severely affect the application's performance.

Hardware-based Trusted Execution Environments (TEEs) are hardware enclaves that protect data and code from the system software. A number of hardware vendors have introduced TEE technologies including Intel with Software Guard Extensions (SGX) [2], [3], ARM with TrustZone [4], MultiZone [5] and others [6], [7], [8], [9], [10]. In TEEs, data can be processed at native speed ensuring that it remains protected even on a third-party machine without having to encrypt it – expensive homomorphic encryption can be avoided to yield better performance.

Despite TEE implementations have been used in a number of industry products [11], [12], programming software that takes advantage of TEE functionalities remains challenging.

Figure 1 shows the implementation of a simple password checker using the C/C++ interface for the Intel SGX enclave (in Microsoft Visual Studio with SGX add-on). With the current approach, programmers need to deal with the low-level details of enclave programming, e.g., partitioning the code into separate files that define the program running outside the enclave (main.cpp) and the program running inside the enclave (enclave.cpp), defining a separate interface between the environments with the semantics of parameter passing (enclave.edl), and setting up the enclave creation (main.cpp, lines 5 to 9) and its disposal after use (main.cpp, line 14).

Though the enclave environment is protected by the hardware, an attacker controlling the non-enclave environment can initiate various attacks on the sensitive data residing inside the enclave, thus compromising the overall application security. In Figure 1, an attacker that controls the non-enclave environment can manipulate the parameters passed to the `checkPassword()` call to the enclave code. In such case, the compiler would not alert the programmer to report a potential security issue.

This leads us to the following key research questions: (a) How to enable seamless integration of enclaves and managed languages like Java? (b) How to check the security of enclave programs with respect to realistic enclave attackers?

In summary, this paper makes the following contributions:

- We present J_E, a language design to seamlessly support enclave programming.
- We describe the implementation of J_E and evaluate its applicability by presenting different case studies.

II. J_E Design

The goal of the J_E design is twofold. (i) The design should abstract away the TEE management details allowing the programmer to easily specify the parts of the program that must run inside the TEE. (ii) The design should provide simple means to specify and enforce security policies for an application. To this end, we provide a set of security annotations and functions. The J_E compiler leverages these annotations to automatically partition the application and generate the logic for the enclave management (creation, initialization, communication). The J_E compiler uses the security annotations and functions to verify information flow policies via a security type system.

enclave.edl

```
enclave {
    trusted {
        public void checkPassword([in, size=len] char* guess, [out]
                int* result, size_t len);
    };
};
```

enclave.cpp

```
const char* password = "secret";
void checkPassword(char* guess, int* result, size_t len) {
    strcmp(guess, password) == 0) ?
        *result = 1 : *result = 0;
}
```

main.cpp

```
#include "sgx_urts.h"
#include "enclave_u.h"
#define BUF_LEN 100
int main() {
    sgx_enclave_id_t eid;
    sgx_status_t ret = SGX_SUCCESS;
    sgx_launch_token_t token = {0};
    int updated = 0;
    ret = sgx_create_enclave(ENCLAVE_FILE, SGX_DEBUG_FLAG, &token,
          &updated, &eid, NULL);
    if (ret != SGX_SUCCESS) { ... /* exception */ }
    char* guess = ... // read guess from stdin
    int result = 0;
    checkPassword(eid, guess, &result, BUF_LEN);
    if (SGX_SUCCESS != sgx_destroy_enclave(eid)) {...}
    return 0;
}
```

Figure 1: Password checker, C++

We illustrate the J_E features using the password checker routine provided in Figure 2. In J_E, a class can be annotated with the `@Enclave` annotation (dubbed *enclave class*). Both code and data of enclave classes are stored inside the enclave. To ensure that data and computations concerning encryption take place within the enclave, the `Password Checker` class in Figure 2 is annotated with the `@Enclave` annotation. Within an enclave class, the `@Secret` annotation identifies *secret* fields. The portions of a program influenced by a secret are also considered secret to prevent flows of sensitive data that may leak outside the enclave. The `password` field (line 3) is annotated with the `@Secret` annotation to denote that its value should not be leaked to the non-enclave environment. The static methods of enclave classes annotated with the `@Gateway` annotation (*gateway* methods) act as the interface between the enclave and the non-enclave environments. The `checkPassword` method (line 6) is annotated with the `@Gateway` annotation. The `checkPassword` method accepts a string from the non-enclave environment and compares it with the `password` field, the result of the comparison is returned to the non-enclave environment as a boolean value. The return value of the gateway methods must not be influenced by secret information.

In addition to annotations, we introduce two operators. The `declassify` is a unary operator to downgrade a secret value into a public one to release sensitive information. The result of the equality comparison of `password` and `guess` is stored in the `result` field (line 8). Since the `result` field is influenced by the `password` secret field, it is also considered as a sensitive. We apply the `declassify` operator to the

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);
} }
```

Figure 2: Password checker, J_E

`result` variable (line 9) to ensure that `result` can be released to the non-enclave environment. The `declassify` operator can only declassify the trusted values. The operator `endorse` endorses an untrusted value into a trusted one. The arguments of gateway methods come from the non-enclave environment and are considered untrusted by default. We apply the `endorse` operator to the `guess` argument (line 7). The trusted value is stored in the variable `guessE`. Hence `result` variable is not influenced by any untrusted value and is declassified successfully (line 9).

III. Attacker Model and Enforcement

In this section we discuss the attacker models considered in J_E, and provide an overview of the security type system used in J_E to enforce security against these attackers.

A. Attacker Model

We assume that the application has two parts – one running inside and the other running outside the enclave. The attacker controls the non-enclave environment by: (1) controlling the non-enclave *data* memory, or (2) controlling the non-enclave *code and data* memory. These attacker capabilities induce two attacker models of interest.

Listing 1 illustrates the attacker models. The program stores a list of secret integers called `secretData`, and provides methods to access single elements of `secretData` and to release the average of these secret integers whenever the trigger `genAvg` is set. In the traditional setting without enclaves, where we trust everything in the system, this program is secure, since the secret values are written to the public variables of the main method only after declassification.

Now, consider a scenario where we need to run this code on an untrusted system. The traditional security assumptions are no longer sufficient, because the attacker can now access the system and learn the `secretData` by simply inspecting the memory.

One way to protect this data on an untrusted system is to use enclaves, thus relying on the hardware features to prevent the attacker from inspecting the enclave memory, and thus, protect the `secretData`. The naive way of achieving

this would be to partition the program in Listing 1 into secret and public parts, and put the `secretData` and all the methods that interact with it in a separate class `Storage` (Listing 2), and put it inside the enclave. The main (public) part of the program remains outside of the enclave (Listing 3).

However, this naive partitioning is not enough to protect the `secretData` stored inside the enclave against different types of attacks from the non-enclave environment. In this work, we investigate two types of attackers that can exploit the enclave−non-enclave interface to learn the secrets stored inside of the enclave.

The first attacker controls the data memory outside of the enclave, hence they can manipulate the parameters passed to `getData` method, and learn all of the elements of `secretData` one by one. The second attacker controls both the data and code memory, hence they can change the order of method calls, e.g., call `Storage.releaseAvg()` in any order, and thus control the release of value `avg`. The enforcement mechanisms implemented in J_E enforce security against these types of active attackers and ensure that enclave programs do not leak secret data.

B. Type System

J_E uses security labels to specify application-level policies. The security labels are not part of the language but are inferred automatically by J_E. A security label is a 2-tuple consisting a confidentiality and an integrity label. We consider two labels *Public* and *Secret* for confidentiality, and two labels *Trusted* and *Untrusted* for integrity. Security labels form a standard (product) security lattice [13] and the order relation among the labels determines the allowed information flows for confidentiality and integrity.

The security type system tracks the implicit and explicit flows of information within the program by checking the security labels at each command, and propagating the security labels accordingly.

The programmer should explicitly specify the data inside the enclave that is considered secret. A secret field is labeled with a *Secret* and *Trusted* security label (2-tuple) as it contains sensitive information originating from inside an enclave class and hence, it is considered not tampered with by an attacker. The rules of the type system prevent storing secret data outside the enclave, prohibit information flow of enclave's secret data to non-enclave environment (unless secret information is intentionally declassified by the programmer in a secure manner), and ensure that gateway methods can only return values having the *Public* confidentiality level.

The type system prevents classes inside of the enclave to call into classes outside of the enclave. This is to control the flow of information and ensure that the only way for passing data to the non-enclave environment is through the return values of gateways.

The type system is mainly standard, but adds some extra safeguards to ensure security against active attackers. To enforce security against *data memory attacker*, we have to make sure that manipulating the parameters of gateway methods does not leak secret data. To achieve this, the type system assigns *Public* and *Untrusted* security label to the data coming from the non-enclave environment, and checks that the declassification of secret data is not influenced by untrusted values, thus ensuring that only the developer controls the decision to release secret data and not the active attacker.

The *data and code memory attacker* is more powerful than the *data memory attacker*. In order to enforce security against this attacker, we have to make sure that changing the order and frequency of calling gateways, or even calling new gateways, does not leak secret data (i.e., it does not lead to declassifying new secrets). To this end, the type system generates a list of all the gateways that declassify secret values and makes sure that all of these gateways are called in all possible executions of the program. This approach ensures that no new declassifying gateways can be called by the active attacker unless it has already been called in some way by the developer. Additionally, to prevent data leaks through changing the order and frequency of gateway calls, the type system marks all of the variables and fields shared between gateways as *Untrusted*. Similar to the parameters of gateway methods, these untrusted values cannot influence declassifications. The formal details of J_E's security type system are presented in [14].

IV. Code Compilation and Implementation

The J_E compilation process involves multiple steps. Figure 4 shows how a J_E program (Listing 4) is partitioned (Listing 5 and 6), translated to Jif to check security (Listing 7 and 8), and augmented with RMI communication (Listing 9 and 10). We now describe these individual compilation steps followed by the implementation details.

Code Partitioning: A J_E program is first analyzed and, based on the annotations, it is split into two partitions – the enclave and the non-enclave partition. All the classes annotated with the `@Enclave` annotation and all their required dependencies belong to the enclave partition. All the remaining classes belong to the non-enclave partition. Listing 4 shows a complete J_E program that encrypts string data using the secret `key` field. The complete program includes a `Main` class (Line 1) and an `Encrypter` class (Line 7). The `Encrypter` class is annotated with the `@Enclave` annotation hence it belongs to the enclave partition. Listing 6 and 5 show the partitioned J_E programs. In this phase, the J_E compiler also performs some correctness checks and collects information required for conversion into an equivalent Jif program (see next section).

Conversion to Jif: Next, the partition to run inside the enclave is converted into an equivalent Jif [15] program. Jif extends Java with security labels to statically enforce information flow control. A Jif security label is a pair consisting of a confidentiality level and an integrity level. In the example, Listing 8 is the equivalent Jif program of Listing 6. The non-enclave partition remains unchanged (Listing 5 and Listing 7). Conversion of the J_E program into Jif involves the following steps. (1) J_E secret fields are

Listing 1: Before partitioning

```
1  class Main {
2     static int[] secretData;
3     static boolean genAvg = false;
4
5     public static void main(String[] args) {
6        int data1 = getData(1);
7        // ...
8        releaseAvg();
9        float avg = getAverage();
10    }
11
12    public static int getData(int input) {
13       return declassify(secretData[input]);
14    }
15
16    public static void releaseAvg() {genAvg = true;}
17
18    public static float getAverage() {
19       if (genAvg) {
20          float avg = doAverage(secretData);
21          return declassify(avg);    }
22       else { return 0.0f; }
23 }  }
```

Listing 2: Inside enclave

```
1  // inside of enclave
2  class Storage {
3     static int[] secretData;
4     static boolean genAvg = false;
5
6     // gateway
7     public static int getData(int input) {
8        return declassify(secretData[input]);
9     }
10
11    // gateway
12    public static void releaseAvg() {genAvg = true;}
13
14    // gateway
15    public static float getAverage() {
16       if (genAvg) {
17          float avg = doAverage(secretData);
18          return declassify(avg);
19       }
20       else { return 0.0f };
21 }  }
```

Listing 3: Outside enclave

```
1  // outside of enclave
2  class Main {
3     public static void main(String[] args) {
4        int data1 = Storage.getData(1);
5        // ...
6        Storage.releaseAvg();
7        float avg = Storage.getAverage();
8  }  }
```

converted into Jif fields with `{Enclave->*;Enclave<-*}` label (Listing 6, Line 3 and Listing 8, Line 2). Such label represents values that are *secret* and *trusted*. (2) For gateway methods, the arguments and the return value are labeled with the Jif's `{}` and `{Enclave->_;Enclave<-*}` labels respectively. The `{}` label denotes *public* and *untrusted* information while the `{Enclave->_;Enclave<-*}` label represents the *public* and *trusted* information (Listing 8, Line 4). (3) The `declassify` operator in J_E corresponds to the `declassify` operator in Jif (Listing 8, Line 7). The obtained Jif program is compiled using the Jif compiler to ensure proper label propagation and checking.

Remote Communication: The next step introduces the code required for the communication via Java RMI [16] between the enclave and the non-enclave partition. As in RMI, remote objects are reachable only through an interface, for each class annotated with the `@Enclave` annotation, the J_E compiler generates a remote interface containing all the gateway methods. Next, the J_E compiler creates a wrapper class implementing the interface above for each enclave class. This way, all the gateway methods of an enclave class are exposed remotely to the non-enclave environment through the remote interface. Finally, the method calls to the enclave class from the non-enclave environment are replaced with an RMI lookup that returns a remote reference to the interface of the wrapper class. Note that, we prohibit remote calls from the enclave to the non-enclave environment. For example, in Listing 10, the J_E compiler generates the `RemoteEncr` interface (Line 2) and the `EncrypterWrapper` class, which acts as a wrapper for the class `Encrypter`, and implements the `RemoteEncr` interface (Line 5). The `EncrypterWrapper` class defines a method `wrapEncrypt` which calls the static method `Encrypter.encrypt` (Line 9). In the non-enclave environment (Listing 9), the J_E compiler transforms the direct calls `Encrypter.encrypt(plaintext)` to a lookup of enclave remote class (Line 6) and to a call the `wrapEncrypt` method of the remote interface `EncrypterWrapper` (Line 8).

Packaging: All the classes to be placed inside and outside the enclave are packaged into two separate executable JAR files. Both JAR files contain an executable class, which includes code for initialization to set up the RMI registry and to publish remote objects required for communication. The user code executes after the initialization phase is complete. The compilation flow is illustrated in figure 3.

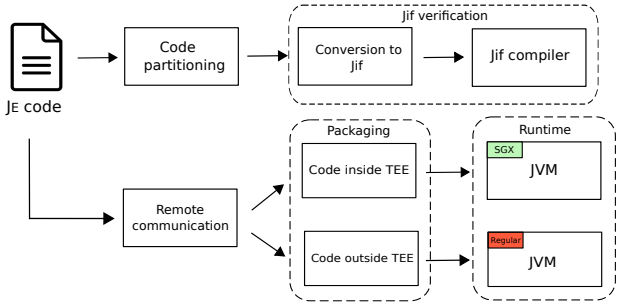

Figure 3: J_E compilation phases

Implementation: We employ JavaParser [17]—a parser library for Java—to perform the code analysis and source code transformation described in section IV. As shown in Figure 3, the enclave program is deployed inside an Intel SGX enclave and executed using a JVM. We use the SGX-LKL framework [18] to support JVM execution inside the enclave. Running a JVM inside the enclave provides the advantages of managed languages but in comparison to running a native code, this approach suffers from a large TCB size.

V. Evaluation

We evaluate J_E with case studies to demonstrate that it can address the security requirements of distributed systems. The aim of the case studies is to demonstrate the core security features of J_E. The case studies consider basic Java constructs, mainly due to Jif's limited Java support and the various static constraints enforced by the Jif compiler. The case studies are of the order of tens of LOC.

We implement a **secure distributed event processing** (CEP) system based on AdaptiveCEP [19]. In CEP, event sources produce events that can carry data and are processed in a cluster of distributed nodes. AdaptiveCEP adapts the

Figure 4: Compilation steps

Listing 4: J_E code

```
class Main {
   public static void main(String[] args) {
      String plaintext = "message";
      String cipher = Encrypter.encrypt(plaintext);
} }
@Enclave
class Encrypter {
   @Secret private static String key;

   @Gateway
   public static String encrypt(String plaintext) {
      String palintextE = endorse plaintext;
      String cipher = encode(plaintext, key);
      return declassify cipher;
} }
```

Listing 5: Partition outside the enclave

```
class Main {
public static void main(String[] args) {
   String plaintext = "message";
   String cipher = Encrypter.encrypt(plaintext);
} }
```

Listing 6: Partition inside the enclave

```
@Enclave
final class Encrypter {
@Secret private static String key;

@Gateway
public static String encrypt(String plaintext) {
   String plaintextE = endorse plaintext;
   String cipher = encode(plaintext, key);
   return declassify cipher;
} }
```

Listing 7: Jif code outside the enclave

```
class Main {
public static void main(String[] args) {
   String plaintext = "message";
   String cipher = Encrypter.encrypt(plaintext);
} }
```

Listing 8: Jif code inside the enclave

```
final class Encrypter [principal Enclave] authority(Enclave) {
private {Enclave->*; Enclave<-*} key;

public String{Enclave->_; Enclave<-*} encrypt{Enclave<-*}(String{}
      plaintext) where authority(Enclave) {
   String plaintextE = endorse(plaintext, {} to {Enclave<-*});
   String{Enclave->*; Enclave<-*} cipher = encode(plaintext, key);
   return declassify (cipher, {Enclave->*; Enclave<-*} to {Enclave->_;
         Enclave<-*});
} }
```

Listing 9: Non-enclave partition – communication

```

class Main{
public static void main(String[] args) {
   String plaintext = "message";

   Remote remoteServer = Naming.lookup("rmi://IPAddr/RemoteEncr");
   RemoteEncr remSrvStub = (RemoteEncr) remoteServer;
   String cipher = remSrvStub.wrapEncrypt(plaintext);
}
```

Listing 10: Enclave partition – communication

```

interface RemoteEncr extends Remote {
   public String wrapEncrypt(String plaintext);
}
class EncrypterWrapper extends UnicastRemoteObject implements
      RemoteEncr {

@Override
public String wrapEncrypt(String plaintext) {
   return Encrypter.encrypt(plaintext);
} }
final class Encrypter { ... }
```

placement of the event processing operators to maximize throughput. In the secure CEP case study, every event is encrypted before it is sent to the cluster. Event processing nodes decrypt the events inside the enclave, perform the processing, and emit the corresponding output events. Listing 11 shows the code for secure CEP using a *filter* event processor node. The `FilterNode` class implements a filter node that can be placed on a remote machine. It contains a gateway method `filter` (Line 20) that accepts an encrypted event and an encrypted predicate. The event is decrypted inside the enclave (Line 21), the predicate is applied to the event payload, and the result is returned (Line 22). This way, the event data are protected from an attacker that controls the OS. The attacker can only observe the encrypted events `EncEvent` passed as an argument to the `process` method.

We also implemented a **secure calculator** as described in [20]. In this case study, confidential data is placed inside the enclave and, during runtime, a user provides tasks to perform on confidential data from the non-enclave environment. The tasks are executed inside the enclave and the result is returned to the non-enclave environment. Listing 12 illustrates tax computation on salary information using the secure calculator. The `TaskProcessor` class is annotated with the `@Enclave` annotation because it contains a secret field `salary` (Line 14) and a gateway method `process` (Line 17) that must be protected within the enclave. A user submits a list of tax-related computation tasks (`taskList`) to the `process` method (Line 4) where the tasks are executed sequentially inside the enclave. The untrusted input `taskList` is first sanitized (Line 18) to verify that it holds certain integrity properties such as it

Listing 11: Secure complex event processing

```
1  class Main   {
2  public static void main(String[] args) {
3     List<IntEvent> events = Generator.getIntEvents(100);
4     List<EncIntEvent> encEvents = encrypt(events);
5     List<EncIntEvent> result = encEvents.stream().
6           filter(PredicateNode.filter).
7           collect(Collectors.toList());
8  }  }
9  class EncIntEvent extends EncEvent {
10    private EncInt val; // encrypted integer
11    private String origin;
12 }
```

```
13 @Enclave
14 class FilterNode  {
15
16 @Secret static String key;
17 Predicate predicate;
18
19 @Gateway
20 public static boolean filter(EncIntEvent event) {
21    Integer val = decrypt(endorse(event).getVal(), key);
22    return apply(predicate, val);
23 }  }
```

is not null and not empty. If the sanitization check succeeds, then the `taskList` field is endorsed (21); otherwise, `null` is returned. When translating to Jif, we use Jif's *checked endorsement* construct to implement input sanitization. The input sanitization reduces the information leakage by only allowing the verified tasks to interact with the secret fields and influence the returned value. The case study demonstrates that the code can be easily partitioned using annotations such that the sensitive information is kept inside the enclave, and a user from the non-enclave environment can only observe the result of predefined sensitive operations that are declassified explicitly by the developer.

Listing 13 shows a J_E implementation of the **Battleship game** as in [21]. In the battleship game, each of the two players owns a secret grid. Initially, players position their battleships randomly on the grid. The game then proceeds in rounds and in each round, players guess a position on the opponent's grid. At the end of a round, players are told if the guessed location contains a battleship. The goal is to successfully guess all the battleship locations on the opponent's grid. The game ends when a player guesses all the battleship locations. This case study demonstrates a scenario where we need to declassify some secret information depending on the external inputs. Initially, both grids are secret and in each round, a grid location needs to be made public via declassification. The location to declassify depends on the guess of the opponent; thus, we need to endorse the opponent's guess to declassify the location. Listing 13 shows the code run on every player's machine. The `Main` class is placed outside the enclave and it stores the status of the opponent's grid (Line 2). The `main` method consists of a while loop in which a player sends and receives guesses. The `Grid` class is placed inside the enclave. The secret variable `grid` (Line 17) stores the status of a player's grid. The gateway method `applyGuess` (Line 20) accepts the opponent's guess as an argument and checks if the guessed location contains a battleship. The argument `guess` is untrusted and we use the `endorse` operator (Line 21) to raise the integrity level to *Trusted*. The `apply` method (Line 25) extracts the array indices from the argument `guess` and checks if a battleship is present at the array location. The variable `result` has *Secret* confidentiality level as it is implicitly influenced by the secret field `grid` (inside the `apply` method, when checking for the battleship location). The subsequent declassification operation (Line 23) downgrades its confidentiality level to *Public*. The declassified value of `result` is returned to non-enclave environment (Line 23). The case study demonstrates the use of the `endorse` and `declassify` operators to declassify secret information securely by endorsing the untrusted values explicitly.

VI. Related Work

In this section we compare J_E to closely related works. We divide them into three broad categories.

Application partitioning for enclaves: Various works have considered partitioning an application for enclaves based on input provided by the user, such as annotations or configuration files. Glamdring [22] performs source-level partitioning of C code based on annotations. Panoply [23] creates low TCB application binaries from the annotated C applications. These works consider input sanitization checks across the enclave interface but do not employ information flow checks. Civet [24] and Uranus [25] perform Java application partitioning based on XML configuration and user annotations respectively. Secure Routines [26] extends annotation-based partitioning for Go programs. Unlike J_E, they consider only passive attackers and provide limited information-flow control guarantees.

Enclaves and information flow control: Gollamudi et al. [20] consider information flow control for enclave applications focusing on erasure policies. DFLATE [27] presents non-interference guarantees in distributed TEEs settings. In contrast to these works, J_E provides robustness guarantees against stronger active attackers. Moat [28] and its successor [29] automate confidentiality verification for enclaves programs.

Information flow control for distributed systems: Various works [30], [31], [32], [33], [34], [35], [36], [37] have employed information flow control techniques to prevent information leaks at the network boundaries in distributed settings. Inspired by these approaches, J_E uses IFC techniques to secure data flow across the enclave - non-enclave interface.

VII. Conclusion

In this paper we presented J_E, a programming framework for enclave-enabled applications where developers use annotations to specify and guide the application partitioning and security policies. We implemented several case studies from literature showing that J_E correctly handles application partitioning while providing strong security guarantees against realistic attackers.

Listing 12: Secure calculator

```
class Main {
   public static void main(String[] args) {
      List<Task> taskList = getTaskSeq("TAX");
      Double tax = TaskProcessor.process(taskList);
   }
}
class Task {
   public Double run (Double input) {
      // Task computation
   }
}
```

```
@Enclave
class TaskProcessor  {
   @Secret static Double salary;

   @Gateway
   public static Double process(List<Task> taskList) {
      if !(sanitize(taskList)) {
         return null;
      }
      List<Task> taskListE = endorse(taskList);
      try {
         Double result = salary;
         for (int i = 0; i < taskListE.size(); i++){
            result = task.run(result);
         }
      } //Omitting the catch block
      return declassify(result);
} }
```

Listing 13: Battleship game

```
class Main   {
   static boolean[][] gridOpp; // opponent's grid status

   public static void main(String[] args) {
      boolean gameOver = false;
      while(!gameOver) {
         Guess g2 = getGuess(); // opponent's guess
         updateOppGrid(g2);
         int result = Grid.applyGuess(g2);
         // generate and send the guess to the opponent
         // along with the result of the previous guess
      }
} }
```

```
@Enclave
class Grid  {

   @Secret private static boolean[][] grid;

   @Gateway
   public static int applyGuess(Guess guess) {
      Guess guessE = endorse(guess);
      int result = apply(guessE); // ship here?
      return declassify(result);
   }
   private static int apply(Guess guess) {
      // check for the presence of battleship
      return result;
} }
```


ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable feedback on this paper. We would like to thank Robert Kubicek, Jonas Seng, and Jesse-Jermaine Richter for their contribution to the J_E implementation. This work is partially supported by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the BRF Project 1025524 from the University of St.Gallen, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

REFERENCES

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, 2018.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," ser. HASP '13, 2013.

[3] Intel, "Intel® software guard extensions developer guide," 2014, accessed 2021-05-20. [Online]. Available: https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-developer-guide.pdf

[4] ARM, "Building a secure system using trustzone® technology," 2009.

[5] S. Pinto and C. Garlati, "Multi zone security for arm cortex-m devices," 2020, https://hex-five.com/wp-content/uploads/2020/02/Multi-Zone-Security-White-Paper-20200224.pdf.

[6] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," 2016, pp. 857–874.

[7] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016.

[8] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *MICRO'52*, 2019, p. 42–56.

[9] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys'20*, 2020.

[10] Apple, "Apple platform security," https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, 2020, accessed 2021-05-20.

[11] Fortanix, "The fortanix runtime encryption," https://fortanix.com/products/runtime-encryption, accessed 2021-05-20.

[12] Anjuna, "Anjuna enterprise enclaves," https://www.anjuna.io/enterprise-enclaves, accessed 2021-05-20.

[13] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, p. 236–243, May 1976.

[14] A. Oak, A. M. Ahmadian, M. Balliu, and G. Salvaneschi, "Language support for secure software development with enclaves," in *CSF 2021*. IEEE, 2021, pp. 1–16.

[15] A. C. Myers, "JFlow: Practical Mostly-static Information Flow Control," in *POPL'99*, 1999, pp. 228–241.

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

[17] JavaParser, "Javaparser," https://javaparser.org, accessed 2021-08-24.

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

[19] P. Weisenburger, M. Luthra, B. Koldehofe, and G. Salvaneschi, "Quality-aware runtime adaptation in complex event processing," in *SEAMS'17*, 2017, p. 140–151.

[20] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *OOPSLA'16*, 2016, p. 494–513.

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

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

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

[24] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An efficient java partitioning framework for hardware enclaves," in *USENIX Security'20*, 2020.

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

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

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

[28] R. Sinha, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *CCS'15*, 2015, pp. 1169–1184.

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

[30] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *SOSP'01*, 2001, p. 1–14.

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

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

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

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

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

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

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