Towards Democratizing Secure Enclave Programming

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*—**
Secure enclaves, like Intel SGX, provide a means to process data securely on third-party cloud infrastructure with little or no performance overhead. Developing software that takes advantage of a secure enclave requires, however, to explicitly deal with a number of low-level details such as dedicated IO, custom syscalls, and stringent memory constraints, requiring the expertise needed for system programming, rather than applications. We discuss our recent research that provides a developer-friendly approach to enclave programming—our Java language extension J_E. Finally, we outline our vision of a programming framework that brings secure enclave programming at the fingertips of application developers.

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

I. Introduction

Cloud computing provides anytime, anyplace computing resources to clients without them facing the burden of managing the computing infrastructure. In this paradigm, a cloud provider offers the cloud infrastructure to the clients on request. This approach ensures on-demand computing and storage provisioning, but it comes at the price of trusting the cloud providers with potentially sensitive data. Hence, the cloud computing paradigm inevitably involves many data security concerns as data is processed on third-party machines that are not under the control of customers. For example, the cloud infrastructure could be controlled by an attacker or it may not have strict access control policies to prevent unauthorized access of data. Traditional privacy-preserving techniques fall short of defending against 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 cloud host. On the other hand, homomorphic encryption schemes [1] allow performing computations directly on 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 other designs like [6], [7], [8], [9] and [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 (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 use various attacks to learn the sensitive data kept 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.

In this invited talk, we discuss (a) Seamless integration of enclaves into managed languages like Java and (b) Security verification of enclave programs with respect to realistic enclave attackers. We present J_E, a language design inspired by our previous work on multitier programming [13], [14], [15], to seamlessly support the development of enclave software. We describe the implementation of J_E and evaluate its applicability by presenting different case studies. Most of the text of this invited talk is taken either verbatim or paraphrased from [16] and [17].

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 perform the necessary steps of enclave management (creation,

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++

initialization, communication). The J_E compiler uses the security annotations and functions to verify information flow policies via a security type system.

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 is returned to the non-enclave environment as a boolean value. The return values of gateway methods must not be influenced by secret information.

In addition to annotations, we provide 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

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

by the `password` secret field, it is also considered as sensitive. We apply the `declassify` operator to the `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

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

Listing 1: Before partitioning

```
class Main {
   static int[] secretData;
   static boolean genAvg = false;

   public static void main(String[] args) {
      int data1 = getData(1);
      // ...
      releaseAvg();
      float avg = getAverage();
   }

   public static int getData(int input) {
      return declassify(secretData[input]);
   }

   public static void releaseAvg() {genAvg = true;}

   public static float getAverage() {
      if (genAvg) {
         float avg = doAverage(secretData);
         return declassify(avg);    }
      else { return 0.0f; }
} }
```

Listing 2: Inside enclave

```
// inside of enclave
class Storage {
   static int[] secretData;
   static boolean genAvg = false;

   // gateway
   public static int getData(int input) {
      return declassify(secretData[input]);
   }

   // gateway
   public static void releaseAvg() {genAvg = true;}

   // gateway
   public static float getAverage() {
      if (genAvg) {
         float avg = doAverage(secretData);
         return declassify(avg);
      }
      else { return 0.0f };
} }
```

Listing 3: Outside enclave

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

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 the enclave: they can manipulate the parameters passed to `getData` method and learn all of the elements of `secretData`. The second attacker controls both the data and code memory: they can change the order of method calls, e.g., call `Storage.releaseAvg()` in any order, and control the release of value `avg`.

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 [18]. 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.

To enforce security against *data memory attacker*, we ensure 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.

For the *data and code memory attacker*, we ensure 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 way, 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 details of J_E's security type system are presented in [16].

IV. Code Compilation and Implementation

The J_E compilation process involves multiple steps.

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.

Conversion to Jif: Next, the partition to run inside the enclave is converted into an equivalent Jif [19] 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. The obtained Jif program is compiled using the Jif compiler to ensure proper label propagation and checking.

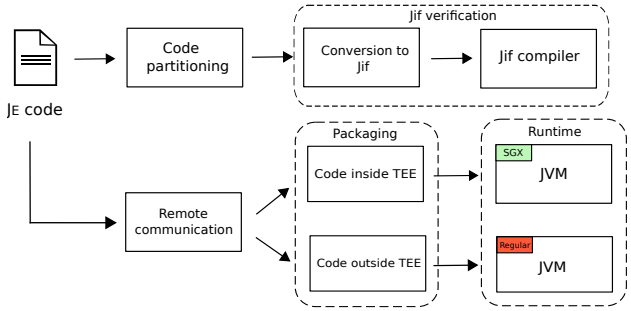

Figure 3: J_E compilation phases

Remote Communication: The next step introduces the communication via Java RMI [20] between the enclave and the non-enclave partition. 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.

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.

Implementation: We employ JavaParser [21] for the code analysis and source code transformation. The enclave program is deployed inside an Intel SGX enclave and executed using a JVM (Figure 3). We use the SGX-LKL framework [22] for the JVM execution inside the enclave.

V. Conclusion

In this invited talk, we presented J_E, a programming framework for enclave-enabled applications that provides language-level abstractions to the developers to specify and guide the application partitioning and security policies. J_E presents a developer-friendly approach to integrating TEE security properties with managed languages.

Acknowledgments

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] P. Weisenburger, J. Wirth, and G. Salvaneschi, "A survey of multitier programming," *ACM Comput. Surv.*, Sep. 2020.
[14] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with scalaloci," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276499
[15] M. Köhler, N. Eskandani, P. Weisenburger, A. Margara, and G. Salvaneschi, "Rethinking safe consistency in distributed object-oriented programming," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 188:1–188:30, Nov. 2020.
[16] 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.
[17] ——, "Enclave-Based Secure Programming with JE," in *2021 IEEE Secure Development*, ser. SecDev '21. Piscataway, NJ, USA: IEEE Press, Oct. 2021.
[18] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, p. 236–243, May 1976.
[19] A. C. Myers, "JFlow: Practical Mostly-static Information Flow Control," in *POPL'99*, 1999, pp. 228–241.
[20] 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.
[21] JavaParser, "Javaparser," https://javaparser.org, accessed 2021-08-24.
[22] 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.