Protecting Security-Sensitive Data Using Program Transformation and Trusted Execution Environment

Anter Abdu Alhag Ali Faree (anterfaree@stu.xidian.edu.cn)
Xidan University School of Electronic Engineering https://orcid.org/0000-0002-9356-3827
Yongzhi Wang
Park University

Research Article

Keywords: Cloud Computing, Confidentiality, Program Partitioning, Program Transformation, Program Analysis, Sensitive data, Trusted Execution Environment TEE, Intel SGX

Posted Date: April 27th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-462176/v1

Protecting security-sensitive data using program transformation and trusted execution environment

Anter Faree[1] and Yongzhi Wang[2]

Abstract

Cloud computing allows clients to upload their sensitive data to the public cloud and perform sensitive computations in those untrusted areas, which drives to possible violations of the confidentiality of client sensitive data. Utilizing Trusted Execution Environments (TEEs) to protect data confidentiality from other software is an effective solution. TEE is supported by different platforms, such as Intel's Software Guard Extension (SGX). SGX provides a TEE, called an enclave, which can be used to protect the integrity of the code and the confidentiality of data. Some efforts have proposed different solutions in order to isolate the execution of security-sensitive code from the rest of the application. Unlike our previous work, CFHider, a hardware-assisted method that aimed to protect only the confidentiality of control flow of applications, in this study, we develop a new approach for partitioning applications into security-sensitive code to be run in the trusted execution setting and cleartext code to be run in the public cloud setting. Our approach leverages program transformation and TEE to hide security-sensitive data of the code. We describe our proposed solution by combining the partitioning technique, program transformation, and TEEs to protect the execution of security-sensitive data of applications. Some former works have shown that most applications can run in their entirety inside trusted areas such as SGX enclaves, and that leads to a large Trusted Computing Base (TCB). Instead, we analyze three case studies, in which we partition real Java applications and employ the SGX enclave to protect the execution of sensitive statements, therefore reducing the TCB. We also showed the advantages of the proposed solution and demonstrated how the confidentiality of security-sensitive data is protected.

Keywords: Cloud Computing, Confidentiality, Program Partitioning, Program Transformation, Program Analysis, Sensitive data, Trusted Execution Environment TEE, Intel SGX.

Introduction

Applications have grown enormously in the public cloud, the total number of applications developed over the cloud has raised intensely over the past few years. However, the security problems that threaten the public cloud are very serious. This threat poses a significant risk to application security in the public cloud. This also has a major impact on application security and privacy [1]. In general, the user's application is required to be uploaded to and performed on the public cloud. However, public clouds are not as enough protected as users imagine. Security violation incidents and vulnerabilities found by researchers [2, 3, 5] appear most commonly. As a result, this can lead to violations of the confidentiality and integrity of security-sensitive data. Revealed incidents including the loss of confidentiality or integrity of data [8, 13] increase these concerns. Under such a circumstance, a key solution to protect cloud users' program confidentiality and integrity in the public cloud setting is required. One of the most important parts of the program is to protect the confidentiality of its sensitive data, which determines the important component in the program that must be protected its data against unintentional, unlawful, or unauthorized access, disclosure, or theft.

To address this concern, server works such as [4, 6] have utilized different Trusted Execution Environments (TEEs) technologies to protect security-sensitive data in applications. In comparison to a cryptographic co-processor, the TEEs are an execution environment from the rest of the applications using the hardware abilities of the platform. Moreover, TEEs protect their data from being accessed from outside the TEEs. The code in TEE is known as a trusted code while the other code is considered an untrusted code. Even though TEE provides security guarantees against strong attacks, few applications employ this technology. One common approach to protect the confidentiality of the security-sensitive data in the code is to annotate some variables (sources) by the developer, which is considered to be very helpful in protecting the confidentiality of the program. Therefore, several studies have investigated protecting data confidentiality in order to achieve a sufficient program's confidentiality protections. The sensitive data in the program including the set of security-sensitive functions, global variables, and local ones are a significant component of the program that needs first to be protected. The work in [7] aimed to extract the control flow and deploy it into a

*Correspondence: ywang@park.edu

[2]Park University, Parkville, MO, USA

Full list of authors' information is available at the end of the article.

trusted environment. Some other efforts on protecting control flow and data flow confidentiality mainly leverage program transformation or distributed architectures. The results from the previous studies have limitations in the aspect of security [7, 9, 51]. Therefore, current works in this track either failed to grant high confidentiality guarantees or incurred high-performance overheads. We claim that our proposed solution is suitable for most TEEs. Despite all those different technologies, we plan to implement the proposed solution based on Intel's Software Guard eXtension (SGX) technology [10, 11] due to its novelty and popularity in its field.

In this paper, we present and analyze our approach based on a fresh direction for securing applications using trusted execution technologies offered by modern CPUs such as SGX. SGX provides a TEE, called an enclave, that protects the integrity of the code and the confidentiality of the data inside it from other software, including the operating system and hypervisor. This paper provides a novel approach to program partitioning for TEE-secured applications. It describes the architecture of the proposed solution and the different phases that lead to the partitioning. In general, our approach can be used for partitioning critical Java applications into security-sensitive code to be run in a trusted execution environment and cleartext code to be run in the public cloud setting. It uses a case study a binary search application to validate the proposed solution. The results of the experimental verification are shown using concrete examples to show how the confidentiality of security-sensitive data is protected.

Our goal in this paper is to propose a security solution that is compatible with all TEE systems and applicable to most Java applications. Our proposed solution analyzes, partitions, and transforms existing Java applications for deployment of the security-sensitive parts and performs the necessary computations in a trusted area such as an SGX enclave.

In general, our proposed solution goes through four main stages as follows.

I. Data Annotation Stage. In this stage, a developer first annotates interest variables in the source code of a Java application that contains security-sensitive data and whose confidentiality should be protected. In other words, the developer provides information about the sources (inputs) of sensitive data by annotating variables whose values must be protected in terms of confidentiality.

II. Data Analysis Stage. Based on the annotation stage, our approach will use static program analysis to find data and control dependencies on security-sensitive data. Our approach will also use static forward slicing to observe a sub-graph with all statements in the program dependence graph (PDG) [12] on which statements in source annotated contain a control and data dependence.

III. Program Partitioning Stage. Based on stages (1) and (2), our solution will generate the partition details (PD) that will define the set of security-sensitive functions and the set of security-sensitive variables. It will also define which part of the code must be placed inside the enclave to protect the confidentiality of its data. PD will also define the transformed program (untrusted code) that will be performed in the public cloud while the sensitive data will be transmitted to an SGX enclave.

IV. Code Generation Stage. In this stage, our solution will demonstrate the computations of the security-sensitive statements inside the enclave based on the output of PD. Moreover, this stage shows how we will return data from the enclave to the user environment. We also show how our approach will react with the security-sensitive and insensitive data that will be deployed to the trusted and untrusted areas, respectively.

Our contributions can be summarized as follows.

- We propose a general solution, that protects the confidentiality of sensitive data on most user-level programs that can be performed on TEE systems such as SGX-supported CPU.
- We analyzed our proposed solution using concrete examples to show how the confidentiality of security-sensitive variables and functions is protected.
- In our case studies, we leverage the program analysis, program partitioning, and SGX technology to hide only the security-sensitive statements of Java code inside an SGX enclave.

Paper Organization

The rest of this paper is organized as follows. In section 2, we give a brief background on TEE systems, SGX technology, and the trusted execution environment. Section 3 introduces the system design of this work. In Section 4, we discuss three case studies that can be applied to our proposed system. Section 5 describes the proposed implementation. In Section 6, we compare the proposed system to the two most related works in the field. Section 7 provides related work. The last section concludes this paper and discusses future work.

Background

Trusted execution environment (TEE)

There are hardware-based solutions such as Intel SGX, ARM TrustZone [40], and software-only approaches, e.g., Virtual Ghost [25] and SKEE [41]. Software-based approaches apply compiler instrumentation or kernel deprivileging to isolate the TEE memory from the kernel memory. TEE provides secure execution of permitted software called Trusted Applications (TAs). The TA is composed of TEE Commands that cooperatively offer secure services to the TA's clients; meanwhile, it forces confidentiality, integrity and access rights

to the code, data, and resources. Each TA is isolated and protected against illegitimate access from other TAs, providing an ecosystem of application vendors.

TEE system such as ARM TrustZone technology offers a system-wide security solution, partitioning the hardware and software resources, therefore, they reside in one of two scenarios, secure scenario for the security subsystem and normal one for everything else. Several Android applications utilize this technology because of the standardization's lack. This subject is addressed by Global Platform [42, 43] that established the standard for managing applications on secure chip technology and a set of specifications for the TEE system architecture.

Intel SGX

Intel SGX grants developers to move their sensitive parts of applications into a protected execution environment, called an enclave, to protect the code confidentiality and data integrity. Code and data of the enclave live in a protected memory region (i.e., the enclave) page cache (EPC). Only the code of application executing inside the enclave is authorized to access the EPC. The confidentiality of enclave memory is secured by transparent memory encryption achieved by the CPU. Enclave calls (ecalls) can be used to enter an enclave and outside calls (ocalls) can be used to call out of the enclave. Therefore, any interaction between the enclave and the OS via system calls, such as network, must execute outside of the enclave. SGX supports local attestation mechanisms which allow an enclave to prove to another enclave that it has a particular digest and runs on the same processor. This privileged mechanism enables the deployment of enclaves that support remote attestation.

In Intel SGX, the size of the TCB contains the enclave code and trusted hardware. Thus, only some portions of an application that require access to sensitive data should be implemented inside the enclave. Some studies [4, 31] have resulted that increasing the code size leads to increasing the number of software bugs. As a result, increasing potential security vulnerabilities. To overcome this problem, it is important to minimize the size of the TCB. However, some factors impact the security of enclave data and code such as the complexity of the enclave interface. For instance, the security-sensitive code inside the enclave needs to interact with the non-enclave environment to call or return some data from/to the enclave.

The Security Model

The security objective is to protect the confidentiality of sensitive statements in the untrusted area, preventing an attacker from reading or modifying the stored sensitive data. To this end, we assume the attackers are interested in obtaining the sensitive data of the program uploaded by the user, i.e., compromising the data confidentiality. However, the attackers are not interested in compromising computation integrity, such as tampering with the computation results. For the environment setting, we assume that the user's zone is free of attacks. However, the public cloud is untrusted. On the public cloud, we assume the processors support SGX. Yet the software stacks on the public cloud host, such as the hypervisor and the OS, are untrusted.

To facilitate our description, we call the enclaves the trusted area and call the software stacks on the public cloud the untrusted area. The attackers can be outside attackers, malicious and cloud vendor employees, or malicious users who are co-hosted with benign cloud users. We do not have special restrictions on the programs to be protected. As long as the program itself does not reveal its sensitive data intentionally (e.g., explicitly printing out the annotated data or other sensitive information), our solution will work well.

System design

Architecture

The architecture of our proposed solution is shown in Fig. 1. For the original Java program P that the user aims to perform on the public cloud, our proposed solution must know which data is security-sensitive in the code of P. Therefore, developers are required to annotate at least one variable in the program (stage 1 in Fig. 1) to provide cues to the partitioning phase. Once the source(s) (variable(s)) are marked in the program, our proposed solution will perform static dataflow analysis (stage 2 in Fig. 1). Based on the output of the dataflow analysis, the proposed solution will use the PDG to performs forward slicing to isolate the security-sensitive data from the code in the program, we call this process the *Partition Details* (*PD*). PD defines which part of the code must be protected by the enclave. In other words, it will partition the original program P into a transformed program PT and the *Sensitive Extracted Data Matrix* (*SEDM*). The latter includes all security-sensitive variables and functions (stage 3 in Fig. 1). After the partitioning, P_T will be uploaded to and performed in the public cloud (i.e., non-enclave area). SEDM will be transmitted to and executed in an SGX enclave. Notice that the user necessarily needs to transmit the SEDM to the enclave in an encrypted manner marked as E(SEDM). In the enclave, we perform necessary computations for all security-sensitive statements inside the enclave based on SEDM (stage 4 in Fig. 1). We provide further discussion about each stage in the following section.

System design

In this section, we present our proposed solution, a new approach for securing applications using TEEs. This solution is built upon hiding the security-sensitive data of applications in terms of code confidentiality. Our approach starts with static dataflow analysis supported by data annotation. Then, we will classify the annotated statements and capture a bunch of the statements that will generate a secure partition to be deployed to an SGX enclave. For the

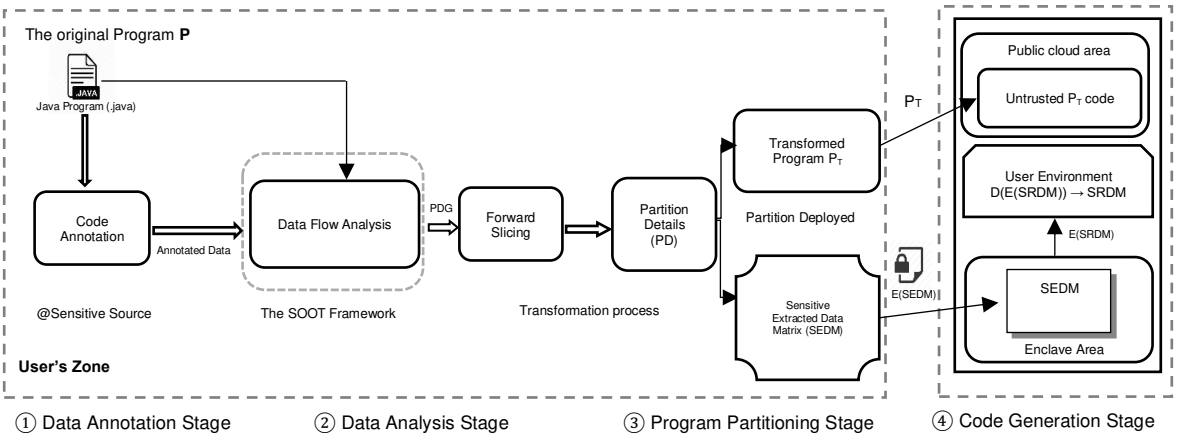

Fig. 1 The Architecture of the proposed solution Engine

partitioning goal, we will define all the statements that deliver confidential data from a certain variable to another one in a given context across reachable paths. To this end, we will apply static dataflow analysis and expand it to accurately capture contextual information by annotating statements that propagate variables from sources to sinks. We will follow standard dataflow analysis algorithms in [44] and [45] to capture sensitive information for a propagating variable statement in a tag $t < source, successor >$, where the source is an incoming security-sensitive variable (predecessor flow), a successor is a security-sensitive variable propagating further (successor flow). The four stages of the proposed solution can be explained as follows.

Data annotation stage

In this stage, our proposed solution must know which variables are security-sensitive in the program. In other words, the developer should provide information about the source(s) of security-sensitive data by annotating variables whose values must be protected in terms of confidentiality. These annotated variables are marked as S_A. To clearly understand how a developer marks security-sensitive data in a program, we consider a piece of Java code in Fig. 2. Only variable x at line 3 is annotated as a security-sensitive variable, indicating that all the variables at line 6,9,11,16,20 and 21 become sensitive variables due to the information flow from the annotated variable x to those statements. Therefore, the annotated variable and all its related security-sensitive statements must be stored and executed inside a special SGX enclave.

Although there is information flow from variable z to c at line 13, it is considered as normal data, because neither variable z nor c has an interaction with the annotated variable x. Meaning that variables z and c are cleartext. Thus, they will be executed in the untrusted area including all other statements that have no interactions with the annotated variable.

```
1.  public class AnotationEx {
2.       public static void main(String args[]) {
3.        @int x = 0; // Sensitive source - Marked by the developer
4.         int y = 4;
5.         int z= 2;
6.         x = y + 3;    // Sensitive statement - Marked by the algorithm
7.         float total = (float) 0.0;
8.         boolean flag;
9.         if (x < y)    // Sensitive statement - Marked by the algorithm
10.         {
11.           total += x;  // Sensitive statement - Marked by the algorithm
12.           flag = true;
13.           return;
14.         }
15.        int c = z++;
16.        int summation = Sum(x , y); // Sensitive statement - Marked by the algorithm
17.        System.out.println("the summation of x and y is:" +summation);
18.         }
19.   public static int Sum(int x, int y) {// Sensitive - Marked by the algorithm
20.         int sum= x+y; // Sensitive statement - Marked by the algorithm
21.         return sum; // Sensitive statement - Marked by the algorithm
22.       }
23.    }
```

Fig. 2 The annotation process and sensitive information flow in a simple Java program.

Data analysis stage

Based on the annotated source(s) code S_A, our approach will distinguish a part of the code that will be considered as a sensitive statement from the one that is considered as insensitive ones. Therefore, the data analysis stage will identify all security-sensitive statements in the program that possess dependencies on the set of all annotated statements S_A. The proposed solution will use static dataflow analysis to examine all security-sensitive statements. Static dataflow analysis is workload independent and therefore conservative decisions must be made about dependencies.

To extract all security-sensitive variables in a Java code, we will transform the original code into another representation. Based on the standard approaches' analysis, we will also use the standard PDG. Where in the standard PDG, vertices represent statements and edges are both data and control dependencies between statements. Therefore, we will use a partition technique mainly based on the graph reachability problem over the PDG. PDGs are considered efficient representations for program partitioning [46]. The program slicing technique was instructed as a sequence of dataflow analysis problems. Using a standard dataflow analysis algorithm and the PDG, our approach will obtain the set of all security-sensitive statements as follows.

Firstly, in term of Static dataflow analysis and by given S_A and PDG, our approach will use graph-reachability to observe a subgraph P_C of PDG which contains all statements with a transitive control or data dependence on statements in PDG (i.e., vertices reachable from statements in S_A via edges in PDG). For statements in S_A that are annotated as security-sensitive data in the program, our approach will use an encryption method [47] to perform encryption on the sensitive data before placing it inside the enclave (i.e., *E(SEDM)*), see Fig 3.

Secondly, given S_A and PDG, our approach will use *static forward slicing* to observe a subgraph P_F with all statements in PDG on which statements in S_A contain a control /data dependence (i.e., all vertices from which statements in S_A are reachable via PDG).

Thirdly, the set of all security-sensitive statements S_T is taken by combining P_C and P_F As a result, our proposed solution constructs a new step, we call this step the partition details PD.

Program partitioning stage

In this stage, we define which part of the code must be placed inside the enclave to protect the confidentiality of data. Based on static program analysis, we will define the code that will be performed in the enclave. As a result, this will define the enclave boundary interface of the sliced code which includes ecall and ocall to the untrusted area. Our approach will construct the partition details (PD) from ST with the set of security-sensitive functions and variables, these sensitive data will be stored in the SEDM in order to transmit them to the enclave area in an encrypted manner. The PD contains all statements' functions and variables that include at least one variable in ST. It also includes the transformed program PT. Moreover, it provides a special function ecall to the non-enclave code to retrieve these security-sensitive variables when needed. Therefore, our proposed solution will generate a tuple for each security-sensitive statement inside the SEDM, marked as L(s).

In General, PD contains two main components; SEDM, which will be deployed to the trusted area (i.e., the enclave), and the transformed program PT that the user aims to execute on the public cloud. The transformation will be achieved at the user's zone inside PD. In the whole process, the insensitive functions and variables remain in the user's zone or the untrusted area. Only the security-sensitive statements in the program will be transmitted to the enclave. As a result, this will create an enclave boundary interface that will establish all security-sensitive statements transmitted to enclave functions and perform all necessary computations inside the enclave and finally return the results outside the enclave (i.e., to the user environment). In general, our proposed solution will check each security-sensitive statement in the SEDM to know whether the statement is a function call, expression statement, or a control flow statement. In other words, we classify each security-sensitive statement in the SEDM into one of the following three types.

Expression statement. For this kind of statement, a tuple will be created, recording some information about that statement. This means, during the transformation process, the proposed solution will replace each security-sensitive expression statement in the program with *bracket* (1), where *bracket* (1) includes two functions, i) the $stmt_{extract()}$ function and ii) the $stmt_{return()}$ function. The function $stmt_{extract()}$ will be used to extract all variables from each sensitive expression statement in the program and store them in the SEDM. For each statement in the $stmt_{extract()}$, a tuple will be created, called $L(stmt)$ represented by the *bracket* (2), records the statement id $stmt_{id}$ that will be used to pick up the proper statement during the execution of the program inside the enclave, statement type st_{type} that will be used to determine the type of statement (either expression or control flow statement), variable type var_{type} that determines the data type of each variable in the security-sensitive statement based on its index in Table 1, the left operand $left_{op}$, the right operand $right_{op}$, and the operator of the statement $stmt_{op}$. The tuple L(*stmt*) can be seen in *bracket* (2).

$$< stmt_{extract(L(stmt),stmt_{id})}, stmt_{return(L(enc_{stmt}),stmt_{id})} > \quad (1)$$

In Table 1, we assume different encoding for each primitive data type in Java (i.e., byte, short, int, long, float, double, boolean, char), plus the object type. After that, we indexed all the data types starting with '00' until '08'. Similarly, these indexes can be used to determine the data type of the returned value by a Java method, for instance, the '09' index can be used when the method does not return a value. In our solution, for each boolean type, we will convert *true* and *false* to 1 and

Table 1 Primitive data types indexes

Data type	*int*	*double*	*float*	*long*	*byte*	*short*	*boolean*	*char*	*object*	*void*
Index	00	01	02	03	04	05	06	07	08	09

0, respectively. We will also use the hash code (an integer value) to represent each object type. The function $stmt_{return()}$ will be used to retrieve the return values of each statement that will be computed inside the enclave based on our scheme in Fig 4. For each statement inside the enclave, there will be return values; those return values will be generated based on the switch-case statement code in each function (i.e., $stmt_{exp}$ function and $stmt_{cf}$ function inside the enclave). A special function will be executed inside the enclave. The idea of the special function is as follows. Based on the statement id $stmt_{id}$ and statement type $stmt_{type}$, it will look up the SEDM, identifying the proper tuple and choose the required variables from L(s) based on the variable type var_{type} and its index in Table 1, and then return the evaluation result of a certain statement to the user environment (see Fig 3). Based on *bracket* (1) and *bracket* (2), the proposed solution will store the sensitive statements of the simple code listed in Fig 2. Table 2 shows the stored sensitive statements of the simple program. The first column in Table 2 represents the statement id that will be generated sequentially for each security-sensitive statement in Fig 2; this means, we will generate a sequence unique number for each security-sensitive statement. The second column represents statement type; for each security-sensitive statement, we use 0 and 1 to assign the expression statement and the control flow statement, respectively. For instance, the statement at line 9 in Fig 2 is a control flow statement, thus we will encode it with 1 as it is shown in Table 2, where the other statements in Fig 2 are expression statements, thus we encode them all with 0. The third column represents the variable type var_{type} which stores the data type of each statement; therefore, we pick up the proper data type from Table 1 based on its definition in Fig 2. The fourth and fifth columns represent the left and right operand for each security-sensitive statement in Fig.2, respectively.

$$< stmt_{id}, stmt_{type}, var_{type}, left_{op}, right_{op}, stmt_{op} > \quad (2)$$

Notice that if the value of the left or the right operand is constant, we will store the actual value in the tuple, otherwise, we will retrieve its position from the corresponding array that will be generated inside the enclave (see Table 5). The last column in Table 2 stores the actual operator of each security-sensitive statement in Fig 2. Table 2 shows all security-sensitive statements in Fig 2, where the $stmt_{id}(0)$, $stmt_{id}(1)$, $stmt_{id}(2)$, $stmt_{id}(3)$, $stmt_{id}(4)$, $stmt_{id}(5)$, $stmt_{id}(6)$, represent lines 3,4,6,7,9, 11 and 14, respectively.

Control flow statement*:* For this kind of statement, we will apply the same solution that will be used in the expression statements above. The control flow statement differs from the expression statement in that the value is stored in the statement type, where 1 and 0 indicate a control flow statement and the expression statement, respectively.

Function call statement: In our proposed solution, we consider a function in a Java program as a security-sensitive function if its body or its definitions contain at least one statement in S_T. In other words, any function that its body or its definitions contain a statement related to the annotated variable(s), will be considered as a sensitive function. For each sensitive function, we replace the sensitive function in a Java application with the *bracket* (3). Note that the function call statement differs from the expression statement and control flow statement in that the stored value in the statement type parameter. Where 2 indicates a function call statement, 0 and 1 indicate the expression statement and control flow statement, respectively.

Where the function $fun_{extract()}$ in *bracket* (3) will be used to extract the statement based on the fun(list) and fun_{id}. Note that the $fun(list)$ is nothing but *bracket* (4). The tuple in *bracket* (4) records the statement *id* ($stmt_{id}$) which defines a

Table 2 The sensitive expression statements and control flow statements of Fig.2 inside the $SEDM$.

$stmt_{id}$	$stmt_{type}$	var_{type}	$left_{op}$	$right_{op}$	$stmt_{op}$
0	00	00	00	*null*	*null*
1	00	00	4	*null*	*null*
2	00	0	01	3	+
3	00	2	00	*null*	*null*
4	01	*null*	00	01	<
5	00	2	20	00	+
6	00	2	20	*null*	*null*

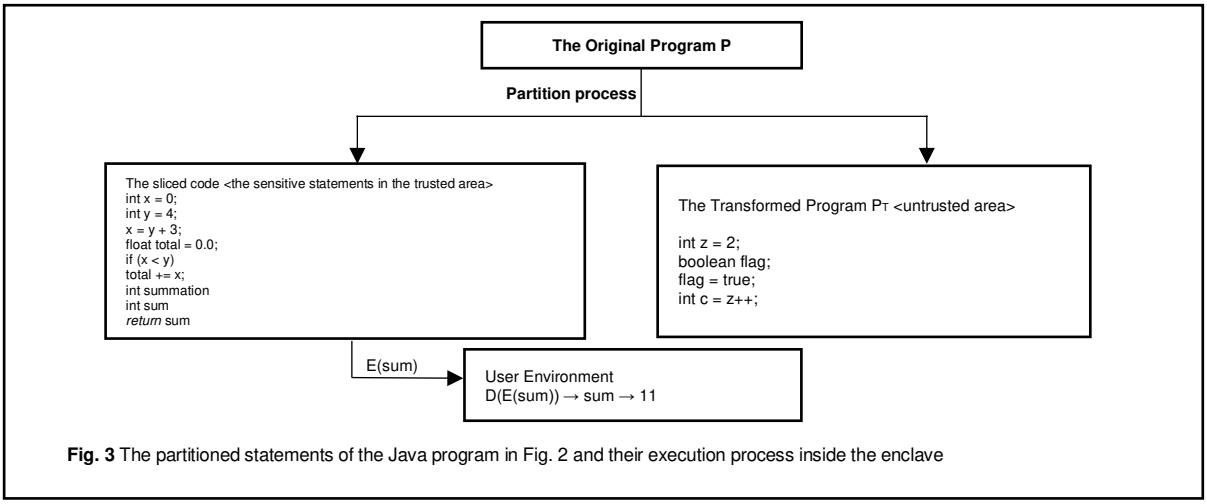

Fig. 3 The partitioned statements of the Java program in Fig. 2 and their execution process inside the enclave

unique Identifier for each security-sensitive statement, the statement type ($stmt_{type}$) which indicates the current statement type, the function id (fun_{id}) states a function modifier ($fun_{modifire}$) defines the access type of unique Identifier for each security-sensitive function, the application), the function name (fun_{name}) which returns the string name function, the function type (fuc_{type}) which returns the return type of the function, and finally the Parameter list ($parm_{(list[])}$) which stores the list of the input parameters, preceded with their data types from the sensitive function and list them in a data matrix as is shown in *bracket* (4) (i.e., from which it can be accessed in Java).

$$< fun_{extract\ (fun(list),fun_{id})} > \quad (3)$$

In our solution, we will index the access modifiers parameter ($fun_{modifire}$) for each sensitive method based on its access modifier in Table 3 and store the index in the enclave. The function $fun_{(return\ ())}$ in *bracket* (5) will be used to read the return values that will be generated inside the enclave for each security-sensitive function based on its statement id and its function *id*. Note that, the list of the return value will be created inside the enclave. For each return value, a tuple will be created in the *Sensitive Returned Data Matrix* (*SRDM*) which can be used to store the returned values from the enclave to the user environment.

$$stmt_{id}, stmt_{type}, fun_{id}, fun_{modifire}, fun_{name,}\ fuc_{type}, parm_{list[]} \quad (4)$$

Meanwhile, we will encrypt the data matrix E(*SRDM*) before we send it to the user environment. In the user environment, we will decrypt the received data matrix D(E(*SRDM*)) during program execution and pick a proper value for each function based on its statement id and function id in the given tuple in the *bracket* (6).

$$< fun_{return_{(fun(enc_{return}))}}, fun_{id} > \quad (5)$$

Table 3 records the return id (ret_{id}), the statement id ($stmt_{id}$), the statement type ($stmt_{type}$), function id (fun_{id}), and

Table 3 The indexes of the access modifiers that will be used in our proposed solution.

Public	Protected	Private	Default
0	1	2	3

the return value of the function (fun_{return}). For the security-sensitive function in the example in Fig 2, we replace line 16 with *bracket* (3). Where the function $fun_{(extract())}$ in *bracket* (3) will be used to extract all information from the target function based on fun(list) (i.e., *bracket* (4)) and fun_{id} and then list all the information in SEDM. The function $fun_{return()}$ in the *bracket* (5) will be used to read the return values that will be generated inside the enclave for each security-sensitive function based on its statement id, statement type, and function *id*. Note that, the list of the return value will be created inside the enclave. For each return value, a tuple will be created in the *Sensitive Returned Data Matrix* (*SRDM*) which can be used to store the returned values from the enclave to the user environment. At that time, we will encrypt the data matrix E(SRDM) before we send it to the user environment. In the user environment, we will decrypt the received data matrix D(E(SRDM)) during program execution and pick a proper value for each function based on its statement id, statement type and function id in the given tuple in the *bracket* (6). Table 4 shows how our proposed solution will store the actual values of the sensitive method at line 16 in Fig 2 in the enclave based on *bracket* (4).

$$< ret_{id}, stmt_{id}, stmt_{type}, fun_{id}, fun_{return} > \quad (6)$$

Code generation stage

The code generation stage demonstrates the whole computations of the security-sensitive statements inside the enclave based on the *Sensitive Extracted Data Matrix (SEDM)*. Moreover, it illustrates the return values that will be transmitted from the enclave to the user environment as is shown in Fig 3. Fig 3 demonstrates the execution process of the security-sensitive and insensitive statements of the program in Fig 2. After the partitioning process, we obtain

Table 4 Storing the actual values of the method *sum* at line 16 in Fig.2 inside the enclave.

$stmt_{id}$	fun_{id}	$stmt_{type}$	$fun_{modifire}$	fun_{name}	fun_{type}	$parm_{list[]}$
7	0	02	0	"Sum"	0	$[x, y] \rightarrow [7,4]$

Table 5 The actual values of the program in Fig.2 inside the enclave.

Sequence	Data type	Position-0	Position -1	Position -2	Position -3
0	int	0	4	11	…
1	double				…
2	long				…

two partitions, the sensitive one on the left side of Fig.3 which includes all security-sensitive variables and functions; and the insensitive part on the right side of Fig 3 which contains all cleartext statements. The statements in the sensitive part will be transmitted to the enclave, where the insensitive ones will be transmitted to the untrusted area. After performing all the necessary computations inside the enclave based on the scheme in Fig 4, the return values will be encrypted inside the enclave using a proper encryption method and returned value to the user environment. We assume that the user environment is a secured area, therefore, we will decrypt the return values in the user environment using a corresponding decryption method to ensure that the returned value will be accessed only by a trusted user and thus cannot be leaked out to the attacker. As it is illustrated in Fig 3, the return value is the function" sum", thus, this value will be encrypted E(*sum*) inside the enclave and then transmitted to the user environment in an encrypted manner.

In the user environment, the return value will be decrypted D(E(sum)) using the same encryption method that is used inside the enclave. The main design scheme of our proposed solution inside the enclave is shown in Fig 4. We define an interface to create several arrays, where each array contains values with the same type and different arrays have different types as is shown in Table 5. We use these arrays to store the actual values of sensitive variables, thus, we only store their positions in the tuples instead of storing the actual values and that is because we aim to secure the actual variables inside the enclave. Therefore, we can read the actual sensitive values using their positions in each matching array. In our scheme, we read these arrays' positions to obtain the actual values of each sensitive value and then perform necessary computations on it. In Fig 4, we show how we will compute

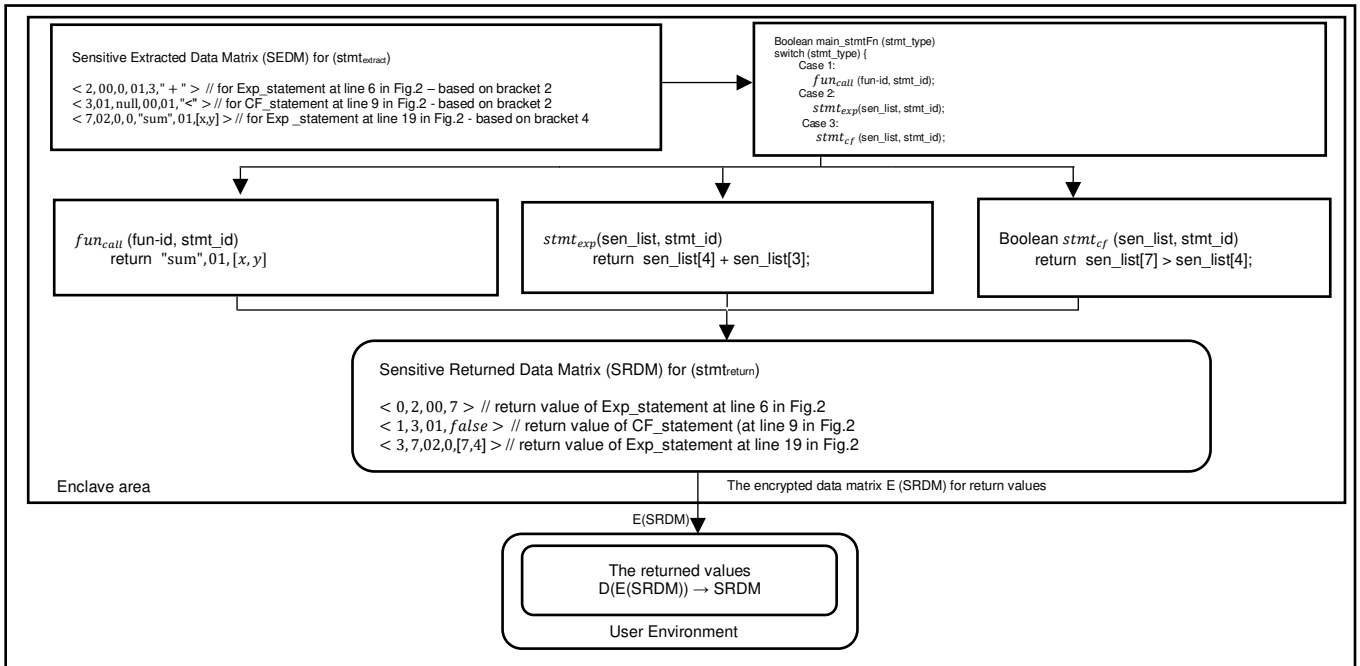

Fig. 4 The scheme of the process execution sequence inside the enclave.

lines 6, 9 and 19 in Fig 2. Moreover, we explore how we achieve the return values for each. Thus, for each executed statement inside the enclave, the return value will be generated and sent back to the user environment.

$$< ret_{id}, stmt_{id}, stmt_{type}, stmt_{ret} > \quad (7)$$

We use the switch-case statement inside the enclave to determine which type of the statement will be executed based on the statement type $stmt_{type}$ (i.e., either expression statement, control flow statement or a function call statement). After each execution, a tuple will be created in (SRDM) based on *bracket* (7) for expression and control flow statements, *bracket* (6) for function call statements, those tuples will be used to store the returned values of each executed statement. The SRDM will be signed and encrypted inside the enclave and sent to the user environment E(SRDM) (see Fig 4).

In the user environment, we will handle the corresponding decryption operations. Thus, we decrypt the received data matrix (i.e., D(E(SRDM)) in the user environment and pick the proper return value for each statement based on the statement id and statement type in the given matrix (SRDM). Notice that both the user environment and enclave use the same encryption method mentioned above. The tuple in *bracket* (7) records the return id ret_{id}, the statement *id* ($stmt_{id}$), statement type $stmt_{type}$, and the return value $stmt_{ret}$ of each expression and control flow returned statement. In our scheme in Fig 4, we define three functions, one for executing function call statements, referred to as fun_{call}, for executing the expression statement, referred to as $stmt_{exp}$ and the other one for executing control flow statements referred to as $stmt_{cf}$. Based on the statement type $stmt_{type}$, the three functions can be invoked from a special function in the enclave, called $main_{stmtFn}$. The difference between the function $stmt_{exp}$ and the function $stmt_{cf()}$ is that in the function $stmt_{exp()}$ we acquire return values that will be returned to the user environment in an encrypted manner, called E(SRDM), where the function $stmt_{cf()}$ " returns a boolean value (either true or false) to determine whether the condition of the control flow statement is executed successfully. In the user environment, we will use the $stmt_{return}$ function in *bracket* (1) with its tuple in *bracket* (7) to retrieve the return values of $stmt_{cf}$ and $stmt_{exp}$ functions from E(SRDM). Similarly, we will use the fun_{return} function in *bracket* (5) with its tuple in *bracket* (6) to retrieve the return values of the fun_{call} function from E(SRDM).

Case study

In this section, we analyze our approach and show its experimental verification by applying it to three real java applications, Binary Search application, Bubble Sort application, and QuickSort application as follows.

Binary Search Application

The binary search application in Fig 5 is a real java application which is a search algorithm that finds the position of a target value within a sorted array.

Data annotation

In this section, we assume the search key at line 19 in Fig 5 is a security-sensitive variable, all other statements that interact with the variable key are security-sensitive statements. Thus, the annotation process at line 19 in Fig 5 marks the content of the variable key as security-sensitive data. Note that the statements in the function *binarysearch*() at lines 4,5,6,7,9,10, and 12 are become security-sensitive statements due to the information flow.

```
1:  class BinarySearch{
2:  public static int binarySearch(int arrBS[], int low, int high, int key)
3:     {
4:           if (high >= low) {
5:           int mid = low + (high - low)/2;
6:           if (arrBS [mid] == key) {
7:           return mid;
8:           }
9:           if (arrBS [mid] > key) {
10:          return binarySearch (arrBS, low, mid-1, key);
11:          } else {
12:          return binarySearch (arrBS, mid+1, high, key);
13:          }
14:     }
15:       return -1;
16:   }
17:  public static void main (String args[]) {
18:    int arrBS[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
19:    @int key = 8; // Sensitive source - Marked by the developer
20:    int last=arrBS.length-1;
21:    int result = binarySearch(arrBS,0, last, key);
22:    if (result == -1)
23:     System.out.println("Element is not exist!");
24:      else
25:   System.out.println("Element is found at index: "+result);
26:     }
27:  }
```

Fig. 5 The Original Binary Search Application.

```
1:  class BinarySearch {
2:  public static int binarySearch(int arrBS[], int low, int high, int key)
3:    {
4:         if (high >= low) {
5:         int mid = low + (high - low)/2;
6:         if (arrBS [mid] == key) {
7:         return mid;
8:         }
9:         if (arrBS [mid] > key) {
10:        return binarySearch (arrBS, low, mid-1, key);
11:        } else {
12:        return binarySearch (arrBS, mid+1, high, key);
13:         }
14: }
15:        return -1;
16:  }
17:  public static void main (String args[]) {
18:    int arrBS[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
19:    @int key = 8;
20:    int last=arrBS.length-1;
21:    int result = binarySearch(arrBS,0, last, key);
22:    if (result == -1)
23:       System.out.println("Element is not exist!");
24:        else
25:  System.out.println("Element is found at index: "+result);
26:        }
27:  }
```

Sliced statements (sensitive statements). Cleartext statements.

Fig. 6 The Partitioned Binary Search Application.

Dataflow analysis

Next, our proposed solution must recognize the annotation variable(s) and PDG. In general, two main steps will be performed as follows.

1. Static dataflow analysis

For analyzing and extracting security-sensitive variables in Fig 5, we will follow standard dataflow analysis algorithms to capture all the sensitive information based on the annotated variable(s) in the program.

2. Static forward slicing

Then, we perform forward slicing to find a subgraph with all statements in PDG on which statements in the variable key (i.e., the annotated variable) have a control and data dependence. As it is shown in Fig 6, the sliced statements are highlighted in yellow colour, while the cleartext ones are highlighted in yellow. We consider the highlighted statements in yellow as sensitive statements, while the ones in yellow are cleartext statements. Fig 5 and Fig 6 illustrate the original binary search and the partitioned one, respectively.

Program partitioning and transformation

In this part, we show how our proposed solution will store sensitive variables that will be trans-mitted to the enclave and the rest of the code that will be deployed to the untrusted area. After performing partitioning on the binary search application, we will perform a transformation process on the code and we target the partitioned part in particular. For each expression statement and control flow statement in the partitioned part, we replace it with *bracket* (1) to extract all the security-sensitive variables. For the function call statement, we replace it with *bracket* (3) to extract all the security-sensitive information. As a result, we store all the security-sensitive variables based on *bracket* (1) and *bracket* (3) in the SEDM. Based on the previous two stages (i.e., data annotation and dataflow), our proposed solution will construct the partition details (PD) which contain security-sensitive variables and insensitive ones. Therefore, we will use the function $stmt_{extract()}$ to extract all variables from each security-sensitive statement in the partitioned part and store them in the SEDM based on *bracket* (2).

Meanwhile, we will store the security-sensitive functions in the SEDM by extracting their data using *bracket* 3(3). Table 6 shows how the security-sensitive statements (the expression statements and control flow statements) in Fig 6 will be stored in the SEDM. Each statement id in Table 6 represents a single sensitive statement in the partitioned code. It also shows all the executed statements of the partitioned code in Fig 6. The $stmt_{id}(0)$, $stmt_{id(1)}$, $stmt_{id}(2)$, $stmt_{id}(3)$, $stmt_{id}(4)$, and $stmt_{id}(5)$ record the expression statement at line 19, the control flow statement at line 4, the expression statements at line 5, the control flow statement at line 6, the expression statements at line 7, and the control flow statement at line 9, respectively. Table 7 shows the information of the binarySearch function in the SEDM.

Code generation

In this section, we show statements' computations of the binary search application that will be executed inside and outside the enclave. As aforementioned in the program partitioning stage, the security-sensitive data will be sent to

Table 6 The sensitive variables of binary search application inside the $SEDM$.

$stmt_{id}$	$stmt_{type}$	var_{type}	$left_{op}$	$right_{op}$	$stmt_{op}$
0	00	0	8	*null*	*null*
1	10	*null*	01	00	> =
2	00	0	02	4	*null*
3	10	*null*	03	00	= =
4	00	0	02	*null*	*null*
5	10	*null*	02	3	>

Table 7 Storing the actual values of the binarySearch method at line 21 in Fig. 6 inside the $SEDM$.

$stmt_{id}$	fun_{id}	$stmt_{type}$	$fun_{modifire}$	fun_{name}	fun_{type}	$parm_{list[]}$
6	0	02	0	"binarySearch"	0	$[arrBS, 0, last, key] \rightarrow [\{3,5,6,8,11,12,14,15,17,18\},0,18,8]$

the enclave side in an encrypted manner (i.e., E(SEDM)). Once the enclave receives the E(SEDM) and verifies the execution environment, it will be able to decrypt D(E(SEDM)) using a corresponding decryption method. Following our security model scheme in Fig 4, we will define an interface in the enclave that will generate several arrays, where each array contains values with the same type and different arrays have different types (see Table 8). For expression and control flow statements, we use these arrays to store actual values of each variable coming from (3) (1) and 2 in the partitioned code, and *bracket* (3) and *bracket* (4) for the functions call statements; next, we will store their positions in tuples instead of storing their actual values. As a result, we will pick up the actual variables using their positions in each array. Table 8 shows that we only have integer values in the binary search application, therefore, all the values will be stored in the integer array (i.e., in sequence 0, first row). In Table 9, *Position*−0, *Position*−1, and *Position*−2 store variables low, high, and mid, respectively. Whereas Position-3 stores the value of the variable key. The positions of the first three variables will keep on updating until we find the required index of the search key element as it appears in Table 9. Table 9 demonstrates the actual values of low, high and mid variables according to their execution sequence inside the enclave.

Eventually, we return the last row in Table 9 to the user environment. To do so, we define three functions inside the enclave (see Fig 4). Based on the statement type $stmt_{type}$, both functions $stmt_{exp()}$ and $stmt_{cf()}$ will be invoked from the main function $main_{stmtFn()}$ and return the proper results. We compute all security-sensitive expression statements and control flow statements in functions $stmt_{exp()}$ and $stmt_{cf()}$, respectively. After performing the necessary computations, we will return all security-sensitive variables to the user environment. In the user environment, we will use the function $stmt_{return()}$ in *bracket* (1) with its tuple in *bracket* (7) to retrieve all expression and control flow values from E(SRDM). Meanwhile, we will use the $fun_{return()}$ function in *bracket* (5) with its tuple in *bracket* (6) to retrieve the function call values from E(SRDM). Fig 8 shows how the proposed solution will hide the security-sensitive function (i.e., the binarySearch method at line 2 and line 21 in Fig 6). Fig 7 shows the three-address code of the security-sensitive method at line 2 in Fig 6 that will be transformed into a new form (i.e., Jimple form). where r0 in Fig 7 refers to the elements of the array *arrBS*[] in Fig 6; *i*0, *i*1, and *i*2 refer to int low, int high, and int key, respectively. The function statement at line 21 will be transformed into a Jimple form as follows:

```
(i2=staticinvoke<BSClass:intbinarySearch
(int[],int,int,int)>(r1, 0, i1, b0);).
```

The above transformation includes the variable result and the function call of the *binarySearch* method at line 21. For the security-sensitive function in the binary search application, we replace it with the *bracket* (3). Where the function $fun_{extract()}$ in *bracket* (3) will be used to extract *fun*(*list*) based on fun_{id}. Notice that the *fun*(*list*) is the *bracket* (4) which contains the statement id $stmt_{id}$, the statement type $stmt_{type}$, the function id fun_{id}, the function modifier $fun_{modifire}$, the function name fun_{name}, the function type fuc_{type}, and finally the parameter list $parm_{list[]}$ to list them all in a data matrix. These values will be placed inside the enclave as it is shown in Table 7. The function $fun_{return()}$ in the *bracket* (5) will be used to read the return values that will be generated inside the enclave for each security-sensitive function based on its statement id and function id. Note that, the list of the return value will be created inside the enclave. For each return value, a tuple will be created in the Sensitive Returned Data Matrix SRDM which will be used to store the returned values from the enclave to the user environment. At that time, we will encrypt the data matrix E(SRDM) before transmitting it to the user environment. In the user environment, we will decrypt the received data matrix D(E(SRDM)) during program execution and pick a proper value for each function based on its statement id and function id in the given tuple in

Table 8 The Actual values of the binary search application inside the enclave.

Sequence	Data type	Position-0	Position -1	Position -2	Position -3
0	int	0	9	4	8
1	double				…
2	float				…
3	long				…

Table 9 The execution sequence of low, high, mid, and key variables inside the enclave.

Statement Execution sequence	Low	High	Mid	Key
1	0	9	4	-
2	0	3	1	-
3	2	3	2	-
4	3	3	3	8

```
1.  public static int binarySearch(int[], int, int, int)
2.  {
4.        int[] r0;
5.        int i0, i1, i2, i3, $i4, $i5, $i6, $i7, $i8, $i9, $i10, $i11;
6.        r0 := @parameter0: int[];
7.        i0 := @parameter1: int;
8.        i1 := @parameter2: int;
9.        i2 := @parameter3: int;
```

Fig. 7 The 3-address code of the sensitive method at line 2 in Fig. 6.

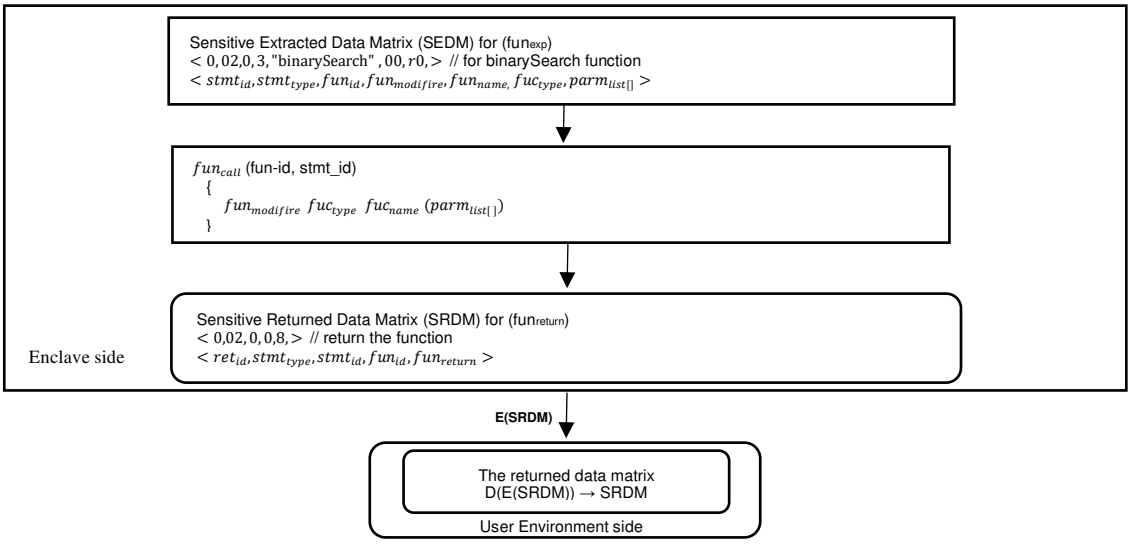

Fig. 8 The process execution sequence of the binary search functions inside and outside the enclave.

the *bracket* (6). The tuple in *bracket* (6) records the return id ret_{id}, the statement id $stmt_{id}$, the statement type $stmt_{type}$, function id fun_{id}, and the return value of the function fun_{return}. This tuple will be used to retrieve all the security-sensitive functions in the program to the user environment.

Bubble sort application

The bubble sort application in Fig 9 is a real java application that is considered as the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

Data annotation

In this subsection, we assume the array arr[] at line 21 in Fig 9 is a security-sensitive statement, all other statements that interact with the array arr[] are security-sensitive statement Thus, the annotation process at line 21 in Fig 9 marks the content of the array arr[]is security-sensitive data. Note that the statements in the function bubbleSort() at lines 2, 3, 4, 5, and 8 are become security-sensitive statements due to the information flow from the annotated statement.

Data flow analysis

Next, our proposed solution must recognize the annotation statement(s) and PDG. In general, two main steps will be performed for this purpose as follows.

1- Static dataflow analysis

For analyzing and extracting the security-sensitive statement in Fig 9, we will follow the standard dataflow analysis algorithm mentioned in the first case study to capture all sensitive information.

2- Static forward slicing

The binary search application in Fig 5 is a real java application which is a search algorithm that finds the position of a target value within a sorted array.

As aforementioned in the program partitioning stage, we perform forward slicing to find a subgraph with all statements in PDG on which statements in the annotated variable have a control and data dependence. As it is shown in Fig 10, the sliced statements are highlighted in yellow, while the cleartext ones are highlighted in yellow. We consider the highlighted statements in yellow as sensitive statements, while the ones in yellow are cleartext statements. Fig 9 and Fig 10 illustrate the original bubble sort and the partitioned one, respectively.

Program partitioning and transformation

In this part, we show how our proposed solution will store sensitive statements that will be transmitted to the enclave and the rest of the code that will be deployed to the untrusted area. After performing the partitioning task on the bubble sort application, we will perform a trans formation process on the code and we target the partitioned part in particular. For each expression statement and control flow statement in the partitioned part, we replace it with *bracket* (1) to extract all the security-sensitive variables. For the function call statement, we replace it with *bracket* (3) to extract all the security-sensitive functions. As a result, we store all the security-sensitive statements based on *bracket* (1) and *bracket* (3) in the SEDM. Based on the two first stages (i.e., data annotation and dataflow), our proposed solution will construct the partition details (PD) which contains security-sensitive variables and insensitive ones.

```
1:  public class BubbleSort{
2:        void bubbleSort(int arr[]) {
3:          int n = arr.length;
4:             for (int i = 0; i < n-1; i++)
5:             for (int j = 0; j < n-i-1; j++)
6:              if (arr[j] > arr[j+1])
7:              {
8:               int temp = arr[j];
9:               arr[j] = arr[j+1];
10:              arr[j+1] = temp;
11:            }
12:         }
13:    void printArray(int arr[]) {
14:      int n = arr.length;
15:       for (int i=0; i<n; ++i)
16:        System.out.print(arr[i] + " ");
17:        System.out.println();
18:     }
19:    public static void main(String args[]) {
20:       BubbleSort ob = new BubbleSort();
21:       *int arr[] = {64, 34, 25, 12, 22, 11, 90};
22:       ob.bubbleSort(arr);
23:       System.out.println("Sorted array");
24:       ob.printArray(arr);
25:       }
26:    }
```

Fig. 9 The Original Bubble Sort Application.

```
1:  public class BubbleSort{
2:       void bubbleSort(int arr[]){
3:         int n = arr.length;
4:            for (int i = 0; i < n-1; i++)
5:            for (int j = 0; j < n-i-1; j++)
6:             if (arr[j] > arr[j+1])
7:             {
8:              int temp = arr[j];
9:              arr[j] = arr[j+1];
10:             arr[j+1] = temp;
11:           }
12:       }
13:    void printArray(int arr[]) {
14:      int n = arr.length;
15:        for (int i=0; i<n; ++i)
16:         System.out.print(arr[i] + " ");
17:        System.out.println();
18:      }
19:    public static void main(String args[]) {
20:       BubbleSort ob = new BubbleSort();
21:       *int arr[] = {64, 34, 25, 12, 22, 11, 90};
22:       ob.bubbleSort(arr);
23:       System.out.println("Sorted array");
24:       ob.printArray(arr);
25:       }
26:    }
```

Sliced statements (sensitive statements). Cleartext statements.

Fig. 10 The Partitioned Bubble Sort Application.

Table 10 The security-sensitive variables of the bubble sort application inside the $SEDM$.

$stmt_{id}$	$stmt_{type}$	var_{type}	$left_{op}$	$right_{op}$	$stmt_{op}$
0	00	0	7	*null*	*null*
1	01	*null*	0	6	<
2	01	*null*	0	7	<
3	01	*null*	6	6	>
4	00	0	0	34	*null*

Therefore, we will use the function $stmt_{extract()}$ to extract all variables from each security-sensitive statement in the partitioned part and store them in the SEDM based on *bracket* (2). Meanwhile, we will store the security-sensitive functions in the SEDM by extracting their data using *bracket* (3). Table 10 shows how the security-sensitive statements (the expression statements and control flow statements) in Fig 9 will be stored in the SEDM. Each statement id in Table 10 represents a single sensitive statement in the partitioned code. It also shows all the executed statements of the partitioned code in Fig 9. The $stmt_{id}(0)$, $stmt_{id}(1)$, $stmt_{id}(2)$, $stmt_{id}(3)$, $stmt_{id}(4)$ record the expression statement at line 3, the control flow statement at line 4, the control flow statement at line 5, and the control flow statement at line 6, the expression statement at line 8, respectively. Table 10 shows the information on the *bubbleSort* function in the SEDM.

Code generation

In this section, we show statements' computations of the bubble sort application that will be executed inside and outside the enclave. As aforementioned in the program partitioning stage, the sensitive data will be transmitted to the enclave side in an encrypted manner (i.e., E(SEDM)). Once the enclave receives the E(SEDM) and verifies the execution environment, it will be able to decrypt D(E(SEDM)) using a corresponding decryption method. Under our security model scheme in Fig 4, we will define an interface in the enclave that will generate several arrays, where each array contains values with the same type and different arrays have different types. For expression and control flow statements, we use these arrays to store actual values of each variable coming from *bracket* (1) and *bracket* (2) in the partitioned code, and *bracket* (3) and *bracket* (4) for the functions call statements;

Table 11 Storing the actual values of the bubbleSort method at line 21 in Fig. 10 inside the $SEDM$.

$stmt_{id}$	fun_{id}	$stmt_{type}$	$fun_{modifire}$	fun_{name}	fun_{type}	$parm_{list[]}$
5	0	02	0	"bubbleSort"	0	$[arr] \rightarrow [11,12,22,25,34,64,90]$ // *sorted array

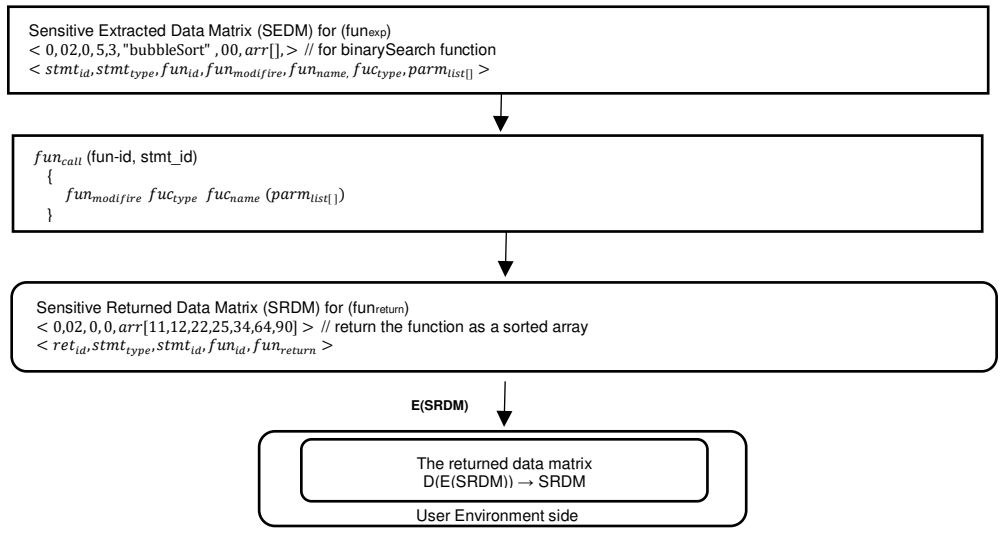

Fig. 11 The process execution sequence of the bubbleSort functions inside and outside the enclave.

```
1:  public static void main(java.lang.String[])
2:  {
3:          java.lang.String[] r0;
4:          testbubblesort.BubbleSort $r1, r2;
5:          int[] r3, $r4;
6:          java.io.PrintStream $r5;
7:          r0 := @parameter0: java.lang.String[];
8:          $r1 = new testbubblesort.BubbleSort;
9:          specialinvoke $r1.<testbubblesort.BubbleSort: void <init>()>();
10:         r2 = $r1;
11:         $r4 = newarray (int)[7];
12:         $r4[0] = 64;
13:         $r4[1] = 34;
14:         $r4[2] = 25;
15:         $r4[3] = 12;
16:         $r4[4] = 22;
17:         $r4[5] = 11;
18:         $r4[6] = 90;
19:         r3 = $r4;
20:         virtualinvoke r2.<testbubblesort.BubbleSort: void bubbleSort(int[])>(r3);
21:         $r5 = <java.lang.System: java.io.PrintStream out>;
22:         virtualinvoke $r5.<java.io.PrintStream: void println(java.lang.String)>("Sorted array");
23:         virtualinvoke r2.<testbubblesort.BubbleSort: void printArray(int[])>(r3);
24:         return;
25:  }
```

Fig. 12 The 3-address code form of the security-sensitive method at line 21 in Fig. 10.

next, we will store their positions in tuples instead of storing their actual values. To return the proper values to the user setting, we define three functions inside the enclave (see Fig 4). Based on the statement type $stmt_{type}$, both functions $stmt_{exp()}$ and $stmt_{cf()}$ will be invoked from the main function $main_{stmtFn()}$ and return the proper results. We will perform all security-sensitive data of expression statements and control fowl statements in functions $stmt_{exp()}$ and $stmt_{cf()}$, respectively. After that, we will return all security-sensitive variables to the user setting. In the user setting, the function $stmt_{return()}$ in *bracket* (1) with its tuple in *bracket* (7) will be used to retrieve all expression and control flow values from E(SRDM). Meanwhile, we will use the $fun_{return()}$ function in *bracket* (5) with its tuple in *bracket* (6) to retrieve the function call values from E(SRDM).

```
1.  public class QuickSort {
2.       public static void main(String[] args) {
3.            int[] x = { 9, 2, 4, 7, 3, 6, 10, 5 };
4.            System.out.println(Arrays.toString(x));
5.            int low = 0;
6.            int high = x.length - 1;
7.            quickSort(x, low, high);
8.            System.out.println(Arrays.toString(x));
9.       }
10. public static void QuickSort(int[] arr, int low, int high) {
11.           if (arr == null || arr.length == 0)
12.                return;
13.           if (low >= high)
14.                return;
15.           int middle = low + (high - low) / 2;
16.           @int pivot = arr[middle];
17.           int i = low, j = high;
18.           while (i <= j) {
19.                while (arr[i] < pivot) {
20.                     i++;
21.                }
22.                while (arr[j] > pivot) {
23.                     j--;
24.                }
25.                if (i <= j) {
26.                     int temp = arr[i];
27.                     arr[i] = arr[j];
28.                     arr[j] = temp;
29.                     i++;
30.                     j--;
31.                }}
32.           if (low < j)
33.                quickSort(arr, low, j);
34.           if (high > i)
35.                quickSort(arr, i, high);
36.      }
37. }
```

Fig. 13 The Original QuickSort Application.

Fig 11 shows how the proposed solution will hide the bubble sort function at line 21 in Fig 10. Fig .12 shows the 3-address code of the security-sensitive method at line 21 in Fig 10 that will be transformed into a Jimple form. The *r*3 in Fig 12 contains the elements of the array arr[] For the security-sensitive function in the bubble sort application, we replace it with the *bracket* (3). Where the function $fun_{extract()}$ in *bracket* (3) will be used to extract fun(list) based on fun_{id}. Notice that the *fun*(*list*) is nothing but the *bracket* (4). These security-sensitive statements will be placed inside the enclave as it is shown in Table 11 for the *bubbleSort* function. The function $fun_{return()}$ in the *bracket* (5) will be used to read the return values that will be generated inside the enclave for each security-sensitive function based on its statement id and function id. Note that, the list of the return values will be created inside the enclave. For each return value, a tuple will be created in the Sensitive SRDM which will be used to store the returned values from the enclave to the user setting. Meanwhile, we will encrypt the data matrix E(SRDM) before transmitting it to the user environment. In the user environment, we will decrypt the received data matrix D(E(SRDM)) during program execution and pick a proper value for each function based on its statement id and its $function_{id}$ in the given tuple in the *bracket* (6). The tuple in *bracket* (6) will be used to retrieve all the security-sensitive functions in the program to the user setting.

Quicksort application

The Quicksort application in Fig 13 is a divide and conquer algorithm. It first divides a large list into two smaller sub-lists and then recursively sorts the two sub-lists. Our proposed solution can be applied to the Quicksort application as follows.

Data annotation

In this section, we assume the variable "*pivot*" at line 16 in Fig 13 is a sensitive variable, and all other statements that interact with the variable "pivot" are security-sensitive statements. Therefore, the annotation process at line 16 in Fig 13 marks the content of the variable "*pivot*" as sensitive data. Likewise, the QuickSort() method at line 10 is considered as a sensitive statement due to the information flow from the annotated variable to the other statements. For the same reason, all statements in the main method at lines 3,4,5,6,7 and 8 are considered sensitive statements.

```
1.  public class QuickSort {
2.       public static void main(String[] args) {
3.            int[] x = { 9, 2, 4, 7, 3, 6, 10, 5 };
4.            System.out.println(Arrays.toString(x));
5.            int low = 0;
6.            int high = x.length - 1;
7.            quickSort(x, low, high);
8.            System.out.println(Arrays.toString(x));
9.       }
10.      public static void QuickSort(int[] arr, int low, int high) {
11.           if (arr == null || arr.length == 0)
12.                return;
13.           if (low >= high)
14.                return;
15.           int middle = low + (high - low) / 2;
16.           @int pivot = arr[middle];
17.           int i = low, j = high;
18.           while (i <= j) {
19.                while (arr[i] < pivot) {
20.                     i++;
21.                }
22.                while (arr[j] > pivot) {
23.                     j--;
24.                }
25.                if (i <= j) {
26.                     int temp = arr[i];
27.                     arr[i] = arr[j];
28.                     arr[j] = temp;
29.                     i++;
30.                     j--;
31.                }
32.           }
33.           if (low < j)
34.                quickSort(arr, low, j);
35.           if (high > i)
36.                quickSort(arr, i, high);
37.      }
38. }
```

Fig. 14 The Partitioned QuickSort Application.

Table 12: Sensitive variables of the QuickSort application inside the SEDM.

$stmt_{id}$	$stmt_{type}$	var_{type}	$left_{op}$	$right_{op}$	$stmt_{op}$
0	00	0	0	7	+
1	00	0	7	null	null
2	00	0	0	null	null
3	00	0	7	null	null
4	01	null	0	7	<
5	01	null	2	7	<

Data flow analysis

Following, our proposed solution will recognize the annotation variable(s) as follows:

1- Static dataflow analysis

For analyzing and extracting sensitive variables in Fig 13, we will utilize the standard dataflow analysis algorithm mentioned in the first case study to extract all sensitive information based on the annotated variable(s) in the program.

2- Static forward slicing

In this phase, we perform forward slicing to find a subgraph with all statements in PDG on which statements in the variable key (i.e., the annotated variable) have a control and data dependence. As it is shown in Fig 14, the sliced statements are highlighted in yellow colour. We consider the highlighted statements in yellow as security-sensitive statements. Fig 13 and Fig 14 illustrate the QuickSort application and the partitioned one, respectively.

Table 13: Storing actual values of the method quickSort in Fig 14 in the SEDM.

$stmt_{id}$	fun_{id}	$stmt_{type}$	$fun_{modifire}$	fun_{name}	fun_{type}	$parm_{(list[])}$
6	0	02	0	”quickSort”	0	[arr[],low,high]ß[2,3,4,5,6,7,9,10,0,7]

```
1:      public static void quickSort(int[], int, int){
2:      int[] r0;
3:      int i0, i1, i2, i3, i4, i5, i6, $i7, $i8, $i9, $i10, $i11, $i12;
4:      r0 := @parameter0: int[];
5:      i0 := @parameter1: int;
6:      i1 := @parameter2: int;
7:      if r0 == null goto label0;
8:      $i7 = lengthof r0;
9:      if $i7 != 0 goto label1;
10:     label0:
11:     return;
12:     label1:
13:     if i0 < i1 goto label2;
14:     return;
15:     label2:
16:     $i8 = i1 - i0;
17:     $i9 = $i8 / 2;
18:     i2 = i0 + $i9;
19:     i3 = r0[i2];
20:     i4 = i0;
21:     i5 = i1;
22:     goto label7;
23:     label3:
24:     i4 = i4 + 1;
25:     label4:
26:     $i10 = r0[i4];
27:     if $i10 < i3 goto label3;
28:     goto label6;
29:     label5:
30:     i5 = i5 + -1;
31:     label6:
32:     $i11 = r0[i5];
33:     if $i11 > i3 goto label5;
34:     if i4 > i5 goto label7;
35:     i6 = r0[i4];
36:     $i12 = r0[i5];
37:     r0[i4] = $i12;
38:     r0[i5] = i6;
39:     i4 = i4 + 1;
40:     i5 = i5 + -1;
41:     label7:
42:     if i4 <= i5 goto label4;
43:     if i0 >= i5 goto label8;
44:     staticinvoke <QuickSort: void quickSort(int[],int,int)>(r0, i0, i5);
45:     label8:
46:     if i1 <= i4 goto label9;
47:     staticinvoke <QuickSort: void quickSort(int[],int,int)>(r0, i4, i1);
48:     label9:
49:     return;
50:     }
```

Fig. 15 The 3-address code of the sensitive method at line 10 in Fig. 14

Program partitioning and transformation

In this phase, we show how our proposed solution will store sensitive variables and the rest of the code that will be deployed to the enclave and the untrusted area, respectively. After performing partitioning on the binary search application, we will perform a transformation process on the code and we target the partitioned part in Fig 14. For each expression statement and control flow statement in the partitioned program, we replace it with the *bracket* (1) to extract all the security-sensitive variables from the code. For the function call statement, we replace it with the *bracket* (3) in order to extract all security-sensitive information from the method itself. Thus, we store all security-sensitive variables that we extracted from the expression statements, control flow statements, and function call statements based on the *bracket* (1) and *bracket* (3) in the SEDM.

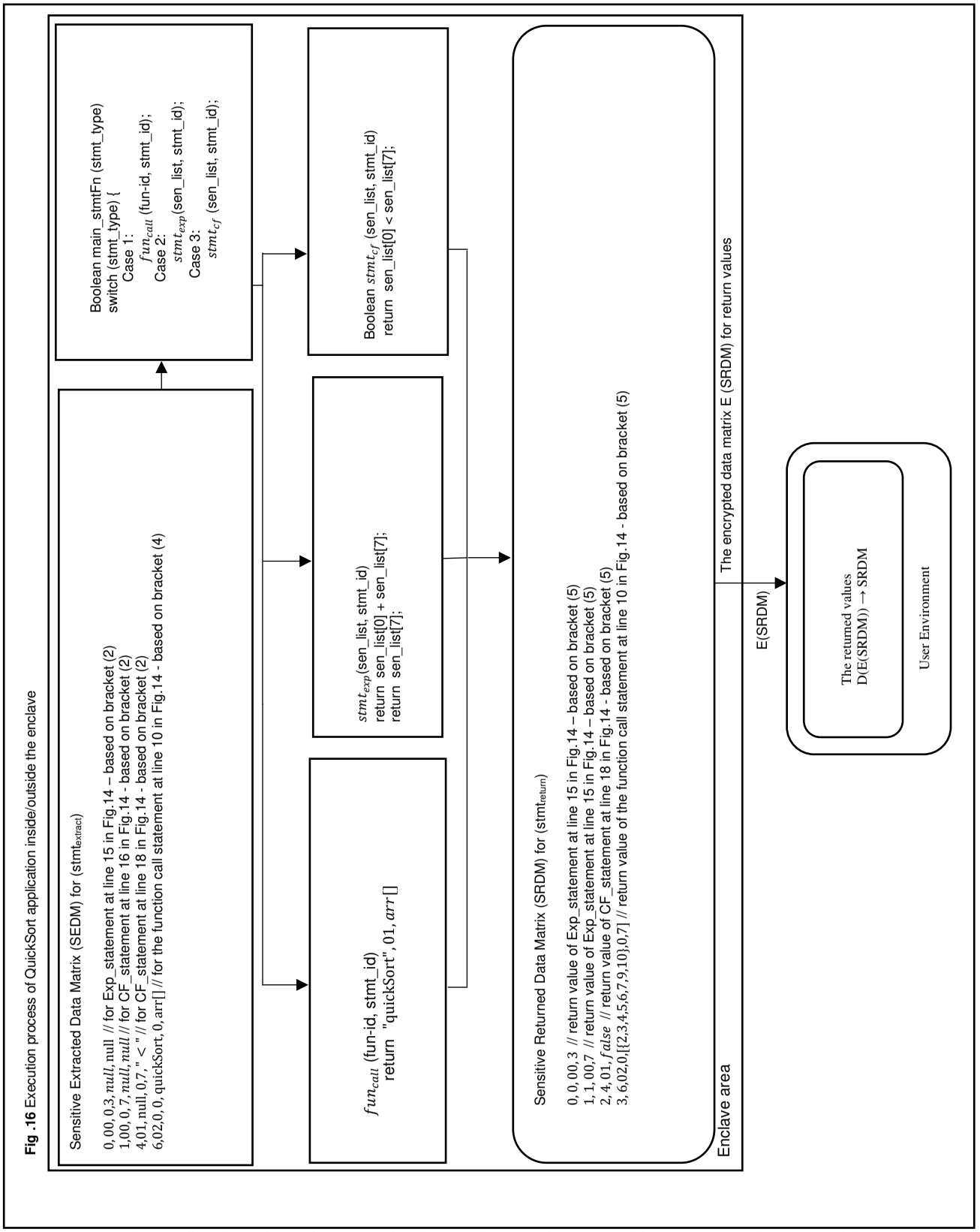

Fig .16 Execution process of QuickSort application inside/outside the enclave

Table 12 records the first iteration of the application and shows how we will store the security-sensitive statements (the expression statements and control flow statements) in Fig 4. in the SEDM. Each statement id in Table 12 represents a single sensitive statement in the partitioned code. It also shows all the executed statements of the partitioned code in Fig 13. The $stmt_{id}(0)$, $stmt_{id}(1)$, $stmt_{id}(2)$, $stmt_{id(3)}$, $stmt_{id(4)}$, and $stmt_{id(5)}$ record the expression statement at line 15, the

expression statement at line 16, the two variables at the expression statements at line 17, the control flow statement at line 18, and the control flow statement at line 19, respectively. Table 11 displays the sorted elements of the" quickSort" method in the SEDM.

Code generation

In this section, we show statements' computations of the QuickSort application that will be executed inside and outside the enclave. As mentioned before in the program partitioning stage, the sensitive data will be transmitted to the enclave side in an encrypted manner (i.e., E(SEDM)). Once the enclave receives the E(SEDM) and verifies the execution environment, it will be able to decrypt the data matrix (i.e., D(E(SEDM)) using the corresponding decryption method. According to the scheme of the security model in Fig 4, we will define an interface in the enclave that will generate several arrays, where each array contains values with the same type and different arrays have different typesFor expression and control flow statements, we use these arrays to store actual values of each variable retrieving from *bracket* (1) and *bracket* (2) in the partitioned code, and *bracket* (3) and *bracket* (4) for the functions call statements. In order to return the proper values to the user setting, we define three functions inside the enclave (see Fig. 4). Based on the statement type $stmt_{type}$, both functions $stmt_{exp()}$ and $stmt_{cf()}$ will be invoked from the main function $main_{stmtFn()}$ and return the proper results. We will perform all security-sensitive of expression statements and control flow statements in functions $stmt_{exp()}$ and $stmt_{cf()}$, respectively. After that, we will return all security-sensitive variables to the user setting. In the user setting, the function $stmt_{return()}$ in *bracket* (1) with its tuple in *bracket* (7) will be used to retrieve all expression and control flow values from E(SRDM). Meanwhile, we will use the $fun_{return()}$ function in *bracket* (5) with its tuple in *bracket* (6) to retrieve the function call values from E(SRDM). Fig 16 shows how the proposed solution will hide the quickSort function at line 10 in Fig 14. The 3-address code of the security-sensitive method at line 10 in Fig 14 is shown in Fig 15. The figure shows the code after transforming it into the Jimple form. The r0 in the statement at line 3 in Fig 15 contains the elements of the array *arr*[]. For the security-sensitive function in the QuickSort application, we replace it with the bracket (3). Where the function $fun_{extract()}$ in *bracket* (3) will be used to extract *fun*(*list*) based on fun_{id}. Notice that the *fun*(*list*) is nothing but the *bracket* (4). For the quickSort method, these security-sensitive statements will be placed inside the enclave as is shown in Table 13. The function $fun_{return()}$ in the bracket (5) will be used to read the return values that will be generated inside the enclave for each security-sensitive function based on its statement id and $function_{id}$. Note that, the list of the return values will be created inside the enclave. For each return value, a tuple will be created in the SRDM which will be used to store the returned values from the enclave to the user setting. Next, we will encrypt the data matrix E(SRDM) before transmitting it to the user environment. In the user environment, we will decrypt the received data matrix D(E(SRDM)) during program execution and pick the proper value for each function based on its statement id and function id in (6). This tuple will be used to retrieve all security-sensitive functions in the program to the user setting. Fig 15 shows the 3-address code of the security-sensitive method at line 10 of the QuickSort application represented in Fig 14. Note that the 3-address code is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Each 3-address instruction has at most three operands and is typically a combination of assignment and a binary operator.

The proposed implementation

In this section, we briefly describe our proposed implementation and validation steps as follows. The workflow of our proposed solution is shown in Fig 1. First, users mark a certain variable(s) as security-sensitive sources in the Java program to be analyzed. We will utilize the FlowDroid [45] to perform this task, we will consider the value tainted by the source as a slicing criterion. Figs 2 and 5 show how developers can mark a certain variable in the code and consider it as a slicing criterion. Once the source(s) will be marked in the program, the Soot framework [48] will be used to analyze the original program and then transform it into another representation (i.e., the 3-address form). The Soot framework is an open-source Java-based compiler tool. The program analysis and transformation can be performed in the Jimple Transformation Pack (*jtp*) phase in the execution of the Soot program. After this step, FlowDroid will be used, a dataflow analysis tool, an extension to the Soot framework to perform static dataflow analysis, and code partitioning. FlowDroid is a static data flow tracker. There is a certain similarity between the two concepts (data slicing and data flow tracker). In our work, we will use the FlowDroid as follows.

FlowDroid generates the main method from the list of entry points. This main method is then used to generate a call graph and an inter-procedural control-flow graph (ICFG). We will then detect all sources which are reachable from the given entry points. Starting at these sources, the taint analysis tracks taints by traversing the ICFG. Thus, the value tainted by the source would be considered as a slicing criterion. FlowDroid will track taints forward through the inter-procedural control flow graph (ICFG). Each statement that transforms a taint abstraction could be seen as part of a code partition. However, since FlowDroid is a taint tracker, it doesn't distinguish between statements that simply pass on taints (because they, e.g., do not reference the tainted value at all) and those that actively transform one taint abstraction into another. We may need to extend the implementation to build a graph of taint-transforming statements while computing the IFDS flow functions. In the end, FlowDroid reports all discovered flows from sources to sinks. Depending on the options the user has chosen either the whole path with all intermediate variables is displayed or

only the source and the sink statement.

The Enclave code will be implemented with Intel SGX SDK. Therefore, all the modules in the Enclave side will be executed in C++. During the implementation, the developer should be aware of and faced with the two main issues of Intel SGX. i) the limited memory size of 128MB plus 4GB (but with huge overhead), we encourage curious readers to solve this issue by using paging support to go beyond that limitation and that is because the limit of 128MB comes from the BIOS itself. Notice that, the Linux driver supports paging, but Windows does not. And ii) the impossibility of execution system calls from within enclaves. The boundary between user and kernel space is system calls. Typically, userspace programs have no direct access to the hardware. Instead, the user space program requests the operating system to allocate memory and perform I/O on its behalf. The system call interface rules the interaction between the operating system and user-space applications. For every system call, an enclave exit and re-entry would be one way to issue system calls in the presence of SGX. Besides the standard user/to transition, the enclave mode switch could be provided. The developer should notice that system calls are disallowed in enclave mode. However, system calls are the standard way for any user-space application to demand service from the privileged operating system kernel. Every valuable program has to allow system calls for external communication; for instance, reading and writing from/to disk and the network involve system calls. We encourage developers to refer to [49] to understand more on how to handle system calls issues.

The modules in the untrusted environment will be executed in Java. The two parts (i.e., the Java side and the C++ side) will be linked with the Java Native Interface (JNI). We will convert some data types in Java into certain C++ types. For instance, we will convert types short, boolean, byte, and object into int type. For each object, we use its hash code (an integer) in C++.

Comparisons

To illustrate the benefit of our proposed solution, we compare it with the most related works in terms of i) system design and ii) theoretical analysis as follows. The system design and theoretical analysis of our proposed solution are inspired by the Glamdring framework [9] and the CFHider prototype system [21]. In the Glamdring framework, it targets C/C++ applications, uses the static analysis function provided by the LLVM compiler to separate the code. Glamdring then automatically partitions the application into untrusted and enclave parts. Glamdring uses data flow analysis to identify functions that may be exposed to sensitive data. It uses backward slicing to identify functions that may affect sensitive data. Glamdring then places security-sensitive functions inside the enclave and adds runtime checks and cryptographic operations at the enclave boundary to protect it from attack.

In CFHider, it protects the control flow confidentiality of the programs and places it in a data matrix to transmit it to the enclave, and then transmit the other part of the program to the untrusted environment. CFHider combines program transformation with Intel SGX. It transforms the condition of each branch statement into a CFQ function call and moves its execution into the enclave that is considered as an opaque and trusted memory space, i.e., the enclave.

However, our proposed solution differs from the aforementioned approaches in that it goes through four stages. The first three stages (Data annotation stage, Data analysis stage, and Program partitioning stage) are inspired by the design of the Glamdring framework. the fourth stage (Code generation stage) is inspired based on the design of the CFHider framework. In our proposed solution, we proposed a prototype system targeting protecting the data confidentiality of Java programs. Our solution will use the static analysis provided by FlowDroid. To perform this task, we will consider the value tainted by the source as a slicing criterion. Once the source(s) will be marked in the program, the Soot framework will be used to analyze the original program and then transform it into another representation (i.e., the 3-address form). Users must first annotate sensitive variable(s). It will partition the original program into a transformed program and the SEDM. The latter includes all security-sensitive variables. After the partitioning, the transformed program will be uploaded to and performed in the public cloud (i.e., non-enclave area). The SEDM will be transmitted to and executed in an SGX enclave. On the enclave side, we will perform necessary computations for all security-sensitive statements inside the enclave based on SEDM.

Table 14: Summary of the comparison between the proposed approach, Glamdring, and CFHider.

Aspects	The Proposed Approach	Glamdring	CFHider
Analysis Mechanism	Forward/Backward data flow analysis	Backward dataflow analysis	Forward dataflow analysis
Protection Type	Data and control flow confidentiality	Data confidentiality and integrity	Control Flow Confidentiality
Programming Languages	Applicable to most PLs	C/C++	Applicable to most PLs
Platforms	Most types of TEE technologies	SGX	SGX

In our future work, we will implement the proposed solution and compare it with some related works in the field concerning performance, evaluation, and execution time.

Table 14 illustrates the comparison between the three approaches. Concerning data flow analysis, our approach applies to both forward and backward data flow analysis, while the Glamdring framework and CFHider prototype are applicable for the backward and forward data flow analysis, respectively. Our approach aims to protect the confidentiality of sensitive data as well as the control flow confidentiality. Glamdring protects the confidentiality and integrity of sensitive data but it cannot protect program control-flow confidentiality. CFHider aims to protect the confidentiality of control flow but not the sensitive data program. Another factor is the programming language, the proposed approach and CFHider are applicable for most programming languages while Glamdring was designed for C/C. The last aspect in our comparison is the platforms that the approaches were designed for. Glamdring and CFHider were designed to be executed in SGX technology. Although we evaluate the proposed solution with SGX in this study, we claim that the proposed solution is suitable for most TEEs.

Related work

TEE Infrastructure

TEEs isolate security-sensitive application logic from the operating system and other applications and therefore protect applications by transmitting confidential partition to TEE. In general, TEEs can be used to decrease the impact of code injection attacks that attempt to steal an application's data, such the case for inaudible data attacks [14-16] or exfiltrate data existing in another TEE.

Some recent efforts [17-19] include general solutions in the standardization of TEE interfaces and protocols. However, most TEEs do not take various types of compartments with various privileges into consideration. PrivateZone [20] presented a framework to enable individual developers to utilize TrustZone resources. In this project, developers can run Security Critical Logics (SCL) in a Private Execution Environment (PrEE). This work relies on ARM TrustZone. ARM servers emerge as a serious and competitive alternative to existing Intel and AMD servers [50].

Program data protection

CFHider [21] and E-CFHider [22] aim to protect the control flow confidentiality in the public cloud setting. However, it hides conditions of branch statements to an opaque SGX enclave and injects fake branch statements to obfuscate the control flow. Yongzhi Wang and Jinpeng Wei [7] proposed runtime control flow obfuscation (RCFO) to protect the confidentiality of the outsourced program control flow. Some existing software-based methods such as [23] and [24] cannot fully meet security, performance, and generality at the same time. These two methods are projected to replace the conditional instructions with lambda calculus and Turing machine simulations, respectively, which can defeat symbolic execution-based reverse-engineering attacks. Virtual Ghost [25] protects application memory from a secured operating system by extending the virtual machine monitor (VMM). These works put trust in the virtual machine monitor, and unable to protect against attackers with privileged access, such as system administrators.

Trusted hardware (SGX)

SGX provides a TEE, called an enclave, that protects the integrity of the code and the confidentiality of the data inside it from other software, including the operating system and hypervisor. LightBox [26] utilizes SGX to build the first system that can drive off-site middleboxes at near-native speed with stateful processing and the most comprehensive protection to data. SGX-Tor [27] presents a practical approach to enhance the security and privacy of Tor by utilizing Intel SGX. EnclaveDB [28] a database engine that guarantees confidentiality, integrity, and freshness for data and queries. Panoply [29] allows applications to be partitioned into multiple compartments and to be run across multiple enclaves following the principle of least privilege. However, this approach is not easily applicable to complex applications such as databases. Oblix [6] a search index for encrypted data that hides access patterns. It relies on a combination of novel oblivious-access techniques and recent hardware SGX enclave platforms. Another study [30] designed a scheme for the existing methods based on software and hardware. Although the scheme was designed based on SGX, it leaks the access pattern.

Graphene [31, 32], and SCONE [4] have verified the possibility of implementing whole applications inside enclaves, supporting that by using appropriate systems, such as a library OS or the C standard library, to the enclave. However, these approaches have a large trusted computing base (TCB) that violates the principle of least privilege due to placing all code inside the enclave. Ryoan [33] aimed to protect the confidentiality of security-sensitive data, it provides a distributed sandbox, leveraging SGX to protect sandbox instances from possibly malicious software. However, it does not protect the confidentiality of program control flow. VC3 [34] the system that lets users running distributed MapReduce computations in the cloud, but placing the code and data in a secure area. VC3 depends on SGX technology to isolate memory regions on individual computers. Bahmani et al. [35] proposed a secure multi-party computation protocol where one of the parties has access to SGX hardware and performs the bulk of the computation. Coppolino Luigi, et al. [36] reviewed some techniques for securing Java software with Intel SGX, the authors selected some promising projects for an experimental comparison in terms of effort, security, and performance. SERECA project [37, 38, 39] aims to remove technical impediments to secure

cloud computing, it proposes to develop a secure environment for reactive cloud applications using Intel SGX.

Conclusions and future work

In this paper, we proposed a solution that can be applied to most TEE systems. Due to the novelty and popularity of Intel SGX in this field, we used SGX technology in our proposed solution as an intended platform to protect the data confidentiality of Java applications.

We describe our proposed solution, the partitioning technique that helps developers leveraging program transformation techniques, program partitioning, and TEE technologies for protecting security-sensitive data of applications. Our proposed solution uses static dataflow analysis to decide which security-sensitive statements must be protected. Therefore, the proposed solution showed how our concrete examples were used to protect their security-sensitive data in terms of confidentiality. Precisely, the proposed solution focuses on protecting the computations of the expression statements, control flow statements, and function call statements of applications in the public cloud setting. The results of the experimental verification are analyzed using real Java concrete applications i.e., Binary Search application, Bubble Sort application and QuickSort application in Fig 2, Fig 5, Fig 6, Fig 9, Fig 10, Fig 13 and Fig 14 to show how the confidentiality of security-sensitive data is protected.

It is our future work to implement the approach, evaluate it, and compare it with other works from the related field. The future work of this research is going to carry out based on the workflow of the proposed approach demonstrated in Fig 1 and the proposed implementation discussed in the proposed implementation section where the necessary steps of the implementation were discussed in detail. In our future work, we will further investigate program analysis mechanisms and partitioning techniques for efficient transformation. However, this proposed solution helps us to obtain a better understanding of how to utilize the program analysis, transformation technique and TEE technologies for protecting security-sensitive data of programs. As a result, it will help us to implement the current proposed approach and thus obtain a much-reduced performance overhead than existing software-based solutions.

Acknowledgement

This paper was supported in part by the National Natural Science Foundation of China (Grant No. 61602364), Key R&D Program of Shaanxi Province (Grant No. 2019ZDLGY12-03, 2019ZDLGY13-06), and the Key Program of NSFC-Tongyong Union Foundation (Grant No. U1636209).

Authors' contributions

The authors equally contributed to this research and the paper initiated and written by the first author.

Authors' information

Anter Faree received his B.S. in Computer Science and Engineering from the National University, Yemen in 2006. He spent six years at the Information Center and Decision Support, Ibb, Yemen as a software developer. He received his MS in Computer Science from Osmania University, India in 2015. He received his Ph.D. in computer science and technology from Xidian University, China, in 2020. His areas of interest include cloud computing security, software security and big data. Contact Anter anterfaree@stu.xidian.edu.cn

Yongzhi Wang received his BE and MS degrees in computer science from Xidian University, China in 2004 and 2007, respectively. He received his Ph.D. in computer science from Florida International University, FL, USA, in 2015. His research interests include software security, cloud computing security, and outsourced computing security. He has published more than 20 peer-reviewed articles. Some of them are published in top-tiered conferences and journals, including INFOCOM, ICSE, CCS, TIFS, IEEE Communications. Contact Wang at ywang@park.edu

Funding

NA

Availability of data and materials

Not Applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1] Xi'an, Shaanxi, China., 710071

[2] Park University, Parkville, MO, USA

References

[1] Yaqoob, I., E. Ahmed, A. Gani, et al., Mobile ad hoc cloud: A survey. 2016. 16(16): p. 2572-2589.

[2] Zhang, A. and X.J.J.o.m.s. Lin, Towards secure and privacy-preserving data sharing in e-health systems via consortium blockchain. 2018. 42(8): p. 140.

[3] Lee, J., J. Jang, Y. Jang, et al. Hacking in darkness: Return-oriented programming against secure enclaves. in 26th USENIX Security Symposium (USENIX Security 17). 2017.

[4] Arnautov, S., B. Trach, F. Gregor, et al. {SCONE}: Secure Linux Containers with Intel SGX. in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 2016.

[5] Weichbrodt, N., A. Kurmus, P. Pietzuch, et al. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. in European Symposium on Research in Computer Security. 2016. Springer.

[6] Mishra, P., R. Poddar, J. Chen, et al. Oblix: An efficient oblivious search index. in 2018 IEEE Symposium on Security and Privacy (SP). 2018. IEEE.

[7] Wang, Y., J.J.C. Wei, Toward protecting control flow confidentiality in cloud-based computation. Computers Security, 2015. 52: p. 106-127.

[8] Bosman, E., K. Razavi, H. Bos, et al. Dedup est machina: Memory deduplication as an advanced exploitation vector. in 2016 IEEE symposium on security and privacy (SP). 2016. IEEE.

[9] Lind, J., C. Priebe, D. Muthukumaran, et al. Glamdring: Automatic Application Partitioning for Intel SGX. in 2017 USENIX Annual Technical Conference (USENIX ATC 17). 2017.

[10] Costan, V. and S. Devadas, Intel SGX Explained. IACR Cryptology ePrint Archive. 2016(086): p. 1-118.

[11] Intel: Intel® Software Guard Extensions Programming Reference (2020).

[12] https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
[13] Ferrante, J., K.J. Ottenstein, J.D.J.A.T.o.P.L. Warren, et al., The program dependence graph and its use in optimization. ACM Transactions on Programming Languages, 1987. 9(3): p. 319-349.
[14] Gruss, D., C. Maurice, and S. Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. 2016. Springer.
[15] Do, Q., B. Martini, and K.K.R. Choo, Is the data on your wearable device secure? An Android Wear smartwatch case study. Software: Practice and Experience, 2017. 47(3): p. 391-403.
[16] D'Orazio, C.J., K.-K.R. Choo, and L.T. Yang, Data exfiltration from Internet of Things devices: iOS devices as case studies. IEEE Internet of Things Journal, 2016. 4(2): p. 524-535.
[17] Do, Q., B. Martini, and K.-K.R. Choo, Exfiltrating data from Android devices. Computers Security, 2015. 48: p. 74-91.
[18] Guan, L., P. Liu, X. Xing, et al. Trustshadow: Secure execution of unmodified applications with arm trustzone. in Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services. 2017.
[19] Ferraiuolo, A., A. Baumann, C. Hawblitzel, et al. Komodo: Using verification to disentangle secure-enclave hardware from software. in Proceedings of the 26th Symposium on Operating Systems Principles. 2017.
[20] Zhao, S., Q. Zhang, Y. Qin, et al. SecTEE: A software-based approach to secure enclave architecture using tee. in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.
[21] Jang, J., C. Choi, J. Lee, et al., Privatezone: Providing a private execution environment using arm trustzone. IEEE Transactions on Dependable Secure Computing, 2016. 15(5): p. 797-810.
[22] Wang, Y., Y. Shen, C. Su, et al. CFHider: Control Flow Obfuscation with Intel SGX. in IEEE INFOCOM 2019-IEEE Conference on Computer Communications. 2019. IEEE.
[23] Zou, Y., Y. Wang, and X. Zhang. Enforcing Control Flow Confidentiality with SGX. in IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). 2020. IEEE.
[24] Lan, P., P. Wang, S. Wang, et al. Lambda obfuscation. in International Conference on Security and Privacy in Communication Systems. 2017. Springer.
[25] Wang, Y., S. Wang, P. Wang, et al. Turing obfuscation. in International Conference on Security and Privacy in Communication Systems. 2017. Springer.
[26] Criswell, J., N. Dautenhahn, and V. Adve, Virtual ghost: Protecting applications from hostile operating systems. ACM SIGARCH Computer Architecture News, 2014. 42(1): p. 81-96.
[27] Duan, H., C. Wang, X. Yuan, et al. LightBox: Full-stack protected stateful middlebox at lightning speed. in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.
[28] Kim, S., J. Han, J. Ha, et al., SGX-Tor: A Secure and Practical Tor Anonymity Network with SGX Enclaves. IEEE/ACM Transactions on Networking, 2018. 26(5): p. 2174-2187.
[29] Priebe, C., K. Vaswani, and M. Costa. Enclavedb: A secure database using SGX. in 2018 IEEE Symposium on Security and Privacy (SP). 2018. IEEE.
[30] Shinde, S., D. Le Tien, S. Tople, et al. Panoply: Low-TCB Linux Applications with SGX Enclaves. in NDSS. 2017.
[31] Xu, J., Y. Zhang, K. Fu, et al., SGX-Based Secure Indexing System. IEEE Access, 2019. 7: p. 77923-77931.
[32] Tsai, C.-C. Graphene Library OS with Intel SGX Support. Available from: https://github.com/oscarlab/graphene.
[33] Tsai, C.-C., K.S. Arora, N. Bandi, et al. Cooperation and security isolation of library OSes for multi-process applications. in Proceedings of the Ninth European Conference on Computer Systems. 2014.
[34] Hunt, T., Z. Zhu, Y. Xu, et al., Ryoan: A distributed sandbox for untrusted computation on secret data. ACM Transactions on Computer Systems, 2018. 35(4): p. 1-32.
[35] Schuster, F., M. Costa, C. Fournet, et al. VC3: Trustworthy data analytics in the cloud using SGX. in 2015 IEEE Symposium on Security and Privacy. 2015. IEEE.
[36] Bahmani, R., M. Barbosa, F. Brasser, et al. Secure multiparty computation from SGX. in International Conference on Financial Cryptography and Data Security. 2017. Springer.
[37] Coppolino, L., S. D'Antonio, G. Mazzeo, et al., A comparative analysis of emerging approaches for securing java software with Intel SGX. Future Generation Computer Systems, 2019. 97: p. 620-633.
[38] Fetzer, C., G. Mazzeo, J. Oliver, et al. Integrating reactive cloud applications in sereca. in Proceedings of the 12th International Conference on Availability, Reliability and Security. 2017.
[39] Brenner, S., T. Hundt, G. Mazzeo, et al. Secure cloud micro services using Intel SGX. in IFIP International Conference on Distributed Applications and Interoperable Systems. 2017. Springer.
[40] SECURE. SECURE ENCLAVES FOR REACTIVE CLOUD APPLICATIONS. 2020. Available from: https://www.serecaproject.eu.
[41] Holdings, A., Building a Secure System using Trust-Zone Technology. 2005, Whitepaper.
[42] Azab, A.M., K. Swidowski, R. Bhutkar, et al. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. in NDSS. 2016.22.
[43] specifications, G.p.d. Global platform device specifications. Available from: https://globalplatform.org/.
[44] Platform, G.P.R.D., Global Platform Device Technology TEE System Architecture. 2011.
[45] Weiser, M., Program slicing. IEEE Transactions on software engineering, 1984(4): p. 352-357.
[46] Arzt, S., S. Rasthofer, C. Fritz, et al., Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices, 2014. 49(6): p. 259-269.
[47] Fritz, C., Flowdroid: A precise and scalable data flow analysis for android. 2013, TU Darmstadt: Darmstadt.
[48] Dworkin, M., Recommendation for block cipher modes of operation. methods and techniques. 2001, National Inst of Standards and Technology Gaithersburg MD Computer security Div.
[49] Padhye, R. and U.P. Khedker. Interprocedural data flow analysis in soot using value contexts. in Proceedings of the 2nd ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. 2013.
[50] Soares, L. and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. in Osdi. 2010.
[51] Hua, Z., J. Gu, Y. Xia, et al. vTZ: Virtualizing ARM TrustZone. in 26th USENIX Security Symposium (USENIX Security 17). 2017.
[52] Faree, A. and Y. Wang. Protecting Security-Sensitive Data Using Program Transformation and Intel SGX. in 2019 International Conference on Networking and Network Applications (NaNA). 2019. IEEE.

Figures

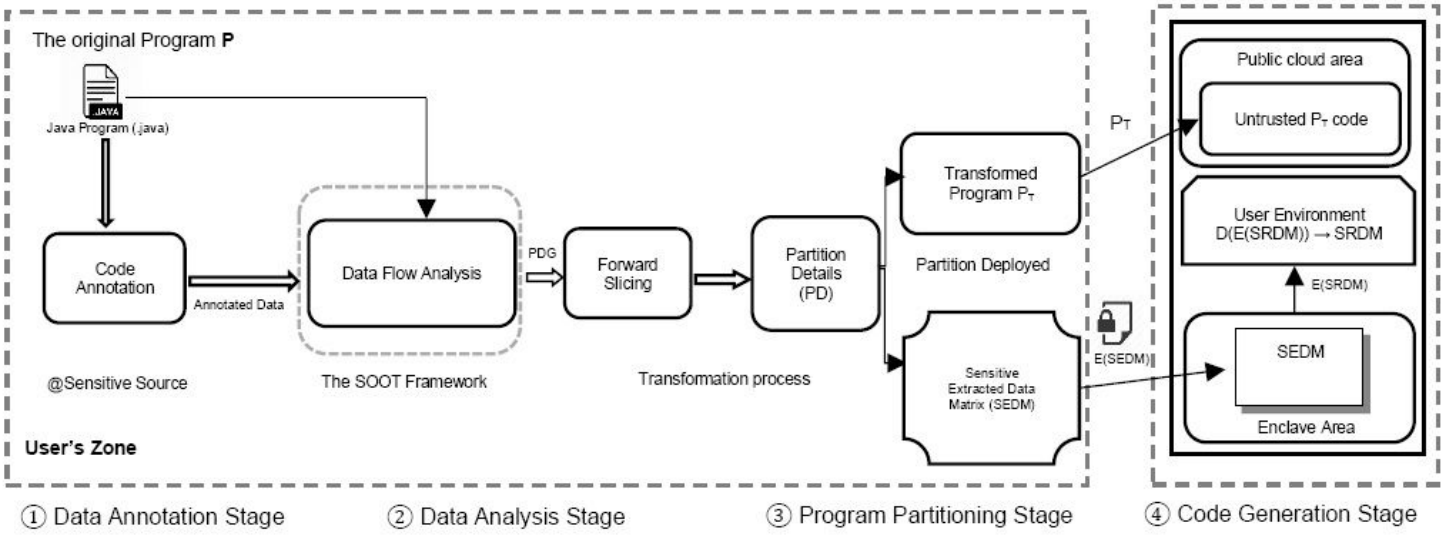

Figure 1

The Architecture of the proposed solution Engine

```
1.  public class AnotationEx {
2.       public static void main(String args[]) {
3.        @int x = 0; // Sensitive source - Marked by the developer
4.         int y = 4;
5.         int z= 2;
6.         x = y + 3;   // Sensitive statement - Marked by the algorithm
7.         float total = (float) 0.0;
8.         boolean flag;
9.         if (x < y)   // Sensitive statement - Marked by the algorithm
10.         {
11.           total += x;  // Sensitive statement - Marked by the algorithm
12.           flag = true;
13.           return;
14.         }
15.        int c = z++;
16.        int summation = Sum(x , y); // Sensitive statement - Marked by the algorithm
17.        System.out.println("the summation of x and y is:" +summation);
18.         }
19.   public static int Sum(int x, int y) {// Sensitive - Marked by the algorithm
20.          int sum= x+y; // Sensitive statement - Marked by the algorithm
21.          return sum; // Sensitive statement - Marked by the algorithm
22.       }
23.    }
```

Figure 2

The annotation process and sensitive information flow in a simple Java program.

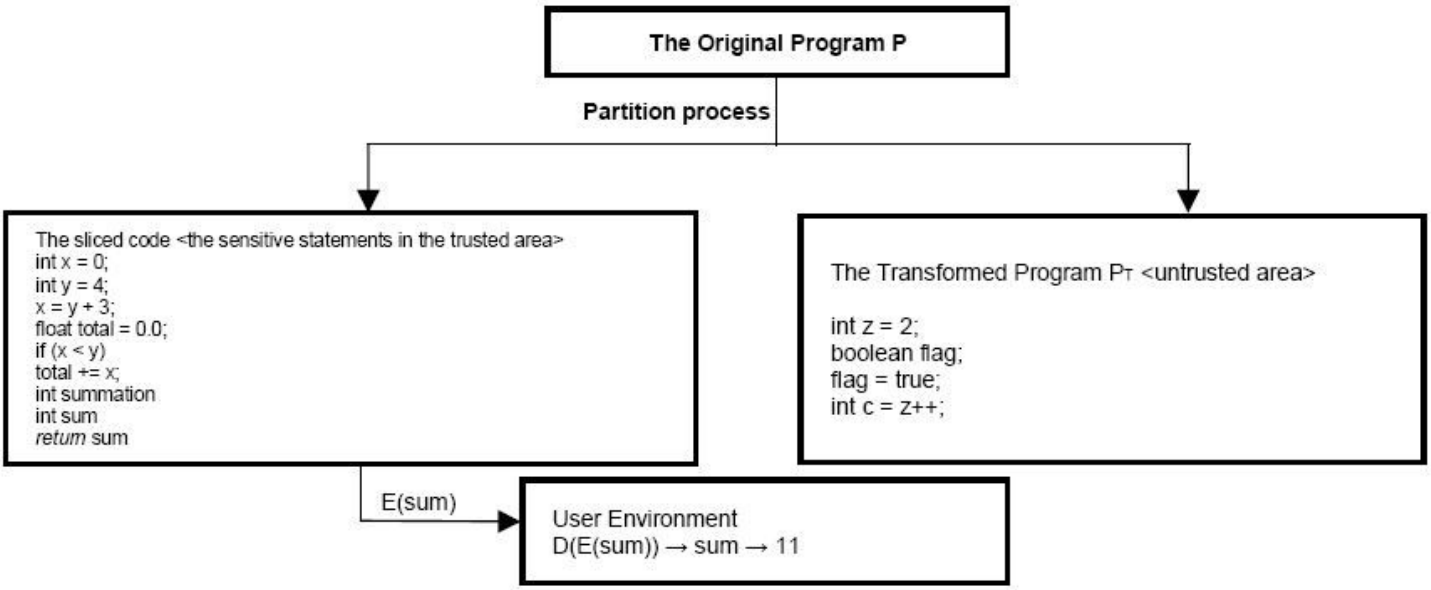

Figure 3

The partitioned statements of the Java program in Fig. 2 and their execution process inside the enclave

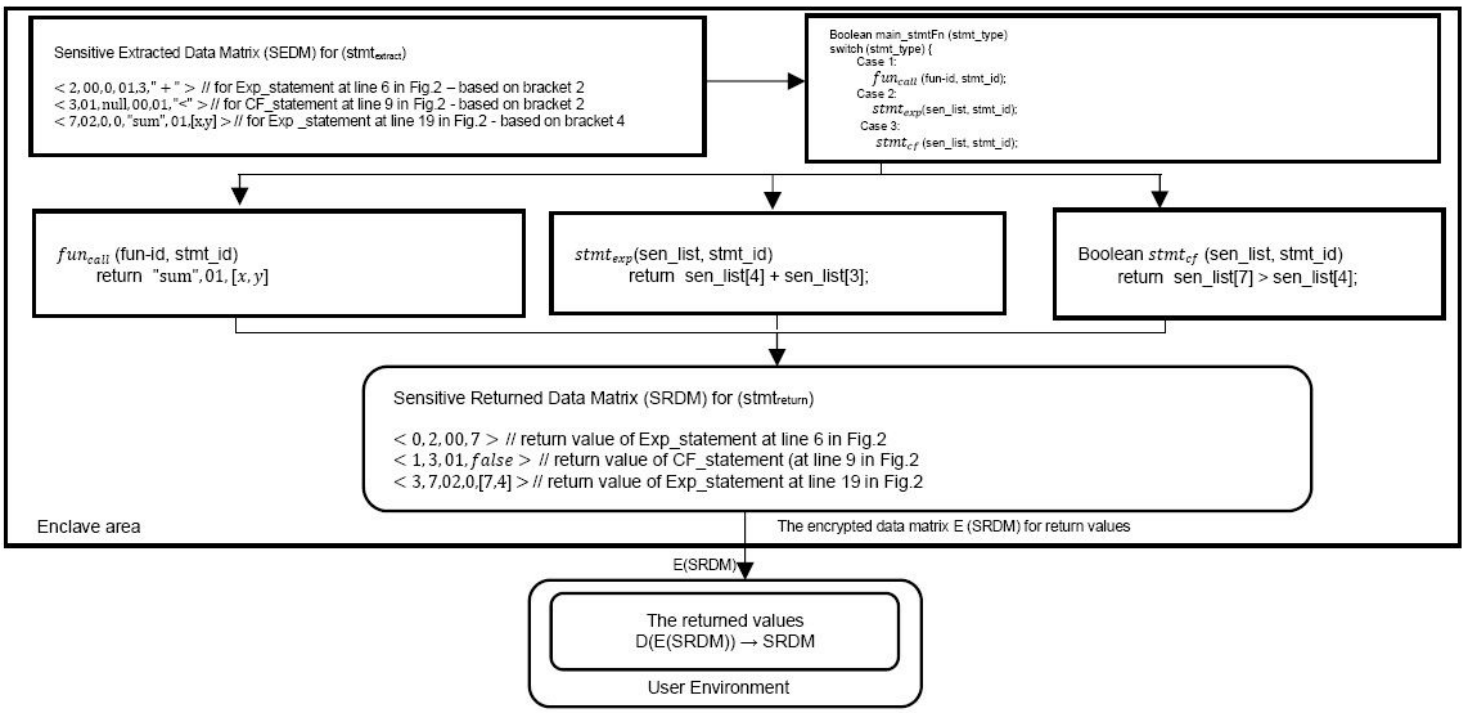

Figure 4

The scheme of the process execution sequence inside the enclave.

```
1:  class BinarySearch{
2:  public static int binarySearch(int arrBS[], int low, int high, int key)
3:     {
4:           if (high >= low) {
5:           int mid = low + (high - low)/2;
6:           if (arrBS [mid] == key) {
7:           return mid;
8:           }
9:           if (arrBS [mid] > key) {
10:          return binarySearch (arrBS, low, mid-1, key);
11:          } else {
12:          return binarySearch (arrBS, mid+1, high, key);
13:          }
14:     }
15:       return -1;
16:   }
17:  public static void main (String args[]) {
18:    int arrBS[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
19:    @int key = 8; // Sensitive source - Marked by the developer
20:    int last=arrBS.length-1;
21:    int result = binarySearch(arrBS,0, last, key);
22:    if (result == -1)
23:     System.out.println("Element is not exist!");
24:      else
25:   System.out.println("Element is found at index: "+result);
26:     }
27:  }
```

Figure 5

The Original Binary Search Application.

```
1:  class BinarySearch {
2:  public static int binarySearch(int arrBS[], int low, int high, int key)
3:    {
4:         if (high >= low) {
5:         int mid = low + (high - low)/2;
6:         if (arrBS [mid] == key) {
7:         return mid;
8:         }
9:         if (arrBS [mid] > key) {
10:        return binarySearch (arrBS, low, mid-1, key);
11:        } else {
12:        return binarySearch (arrBS, mid+1, high, key);
13:         }
14: }
15:        return -1;
16:  }
17:  public static void main (String args[]) {
18:    int arrBS[] = {3, 5, 6, 8, 11, 12, 14, 15, 17, 18};
19:    @int key = 8;
20:    int last=arrBS.length-1;
21:    int result = binarySearch(arrBS,0, last, key);
22:    if (result == -1)
23:       System.out.println("Element is not exist!");
24:        else
25:  System.out.println("Element is found at index: "+result);
26:        }
27:  }
```

Sliced statements (sensitive statements). Cleartext statements.

Figure 6

The Partitioned Binary Search Application.

```
1.  public static int binarySearch(int[], int, int, int)
2.  {
4.        int[] r0;
5.        int i0, i1, i2, i3, $i4, $i5, $i6, $i7, $i8, $i9, $i10, $i11;
6.        r0 := @parameter0: int[];
7.        i0 := @parameter1: int;
8.        i1 := @parameter2: int;
9.        i2 := @parameter3: int;
```

Figure 7

The 3-address code of the sensitive method at line 2 in Fig. 6.

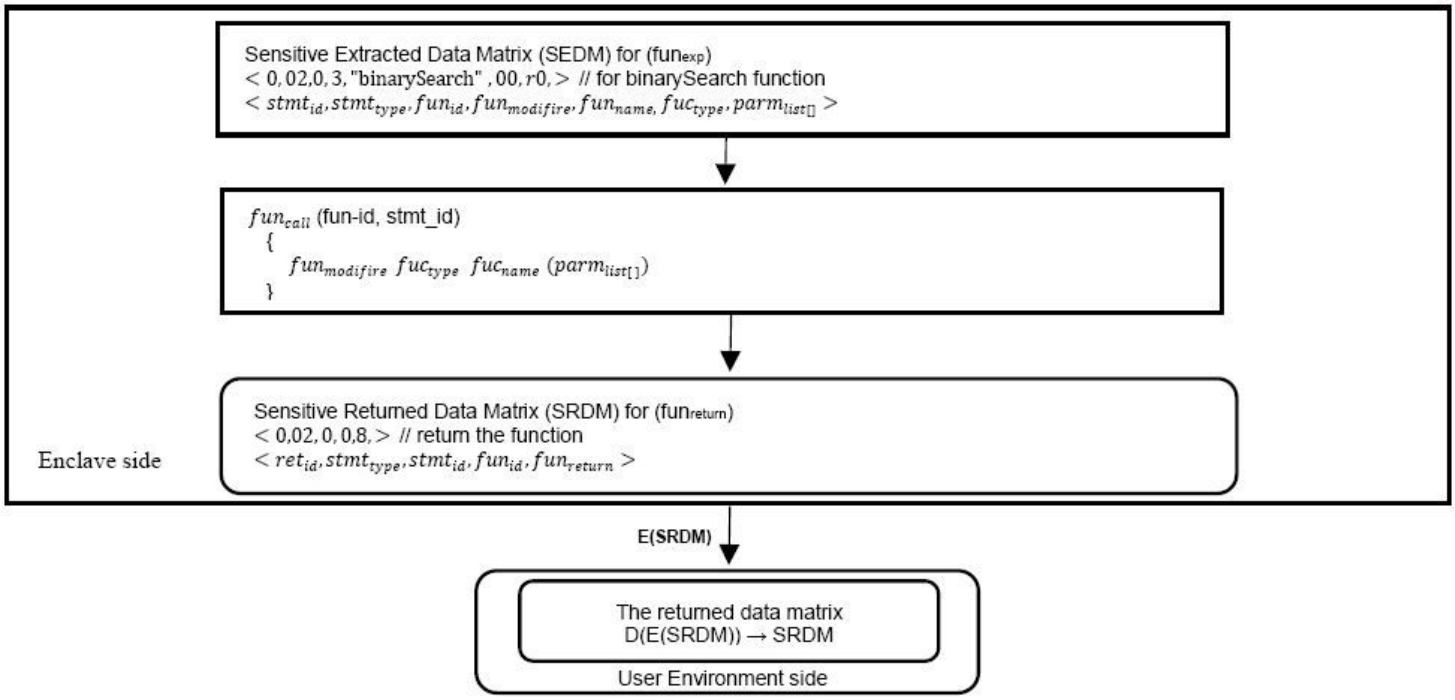

Figure 8

The process execution sequence of the binary search functions inside and outside the enclave.

```
1:  public class BubbleSort{
2:        void bubbleSort(int arr[]) {
3:          int n = arr.length;
4:             for (int i = 0; i < n-1; i++)
5:             for (int j = 0; j < n-i-1; j++)
6:              if (arr[j] > arr[j+1])
7:              {
8:               int temp = arr[j];
9:               arr[j] = arr[j+1];
10:              arr[j+1] = temp;
11:            }
12:         }
13:    void printArray(int arr[]) {
14:      int n = arr.length;
15:        for (int i=0; i<n; ++i)
16:         System.out.print(arr[i] + " ");
17:         System.out.println();
18:     }
19:    public static void main(String args[]) {
20:       BubbleSort ob = new BubbleSort();
21:       *int arr[] = {64, 34, 25, 12, 22, 11, 90};
22:       ob.bubbleSort(arr);
23:       System.out.println("Sorted array");
24:       ob.printArray(arr);
25:       }
26:    }
```

Figure 9

The Original Bubble Sort Application.

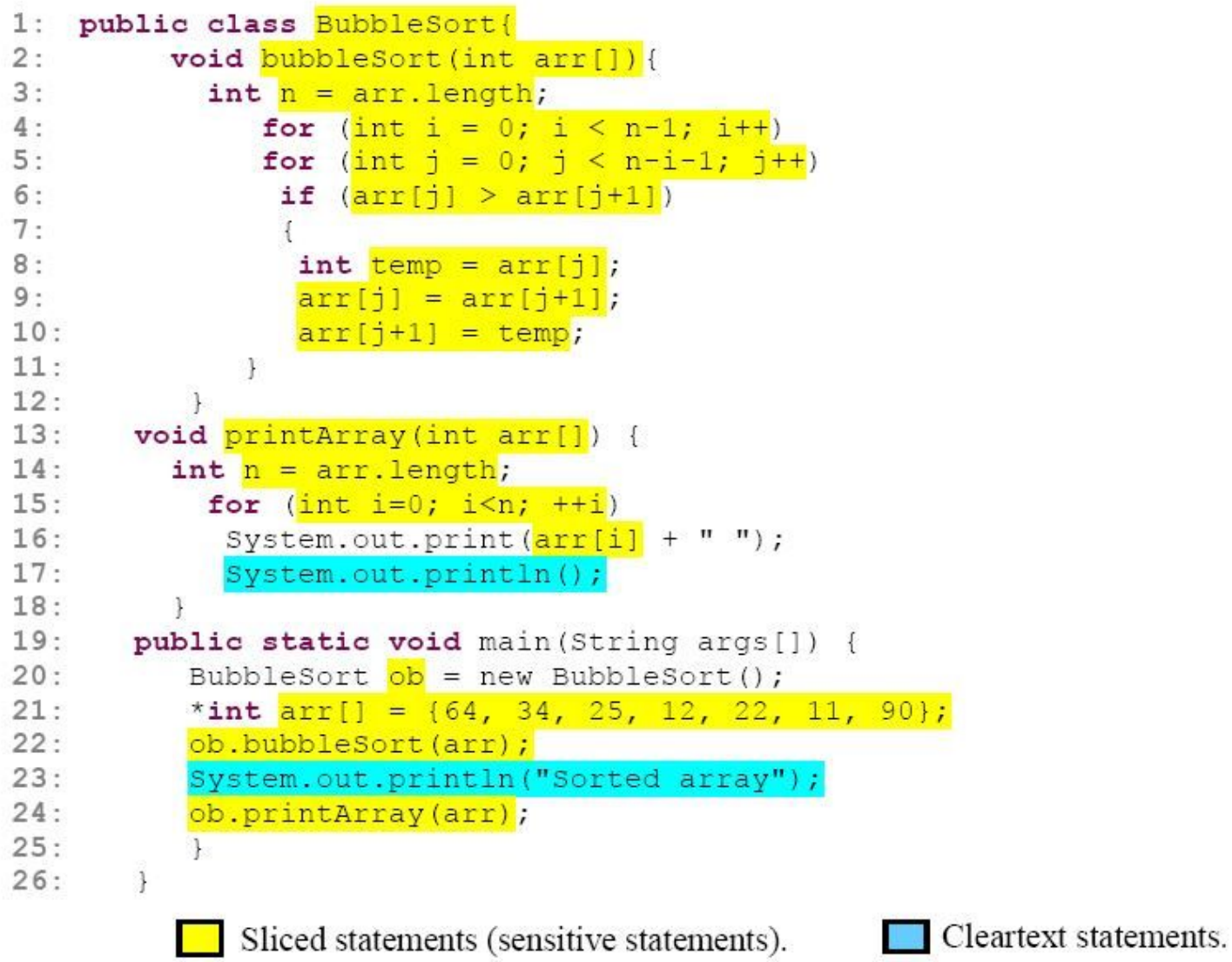

Figure 10

The Partitioned Bubble Sort Application.

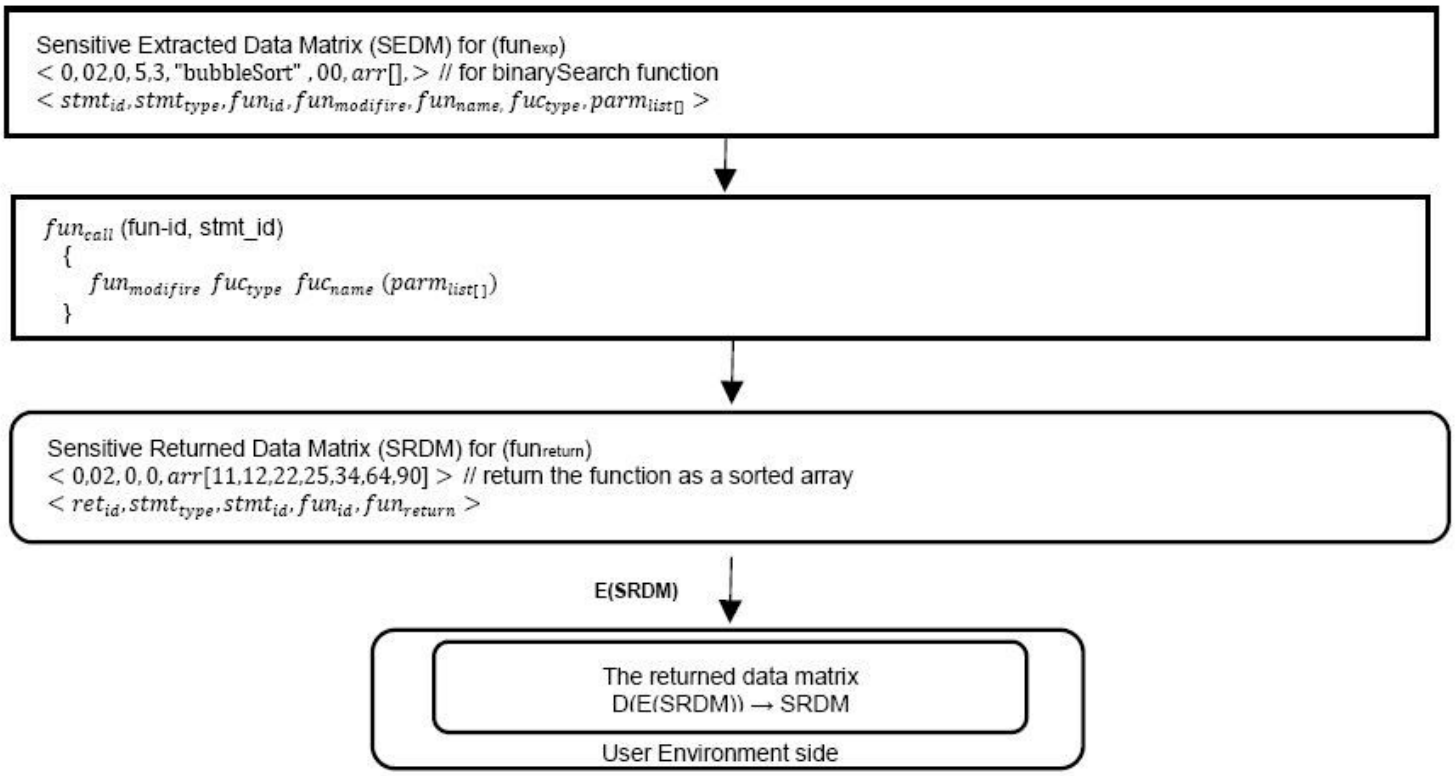

Figure 11

The process execution sequence of the bubble Sort functions inside and outside the enclave.

```
1:  public static void main(java.lang.String[])
2:  {
3:          java.lang.String[] r0;
4:          testbubblesort.BubbleSort $r1, r2;
5:          int[] r3, $r4;
6:          java.io.PrintStream $r5;
7:          r0 := @parameter0: java.lang.String[];
8:          $r1 = new testbubblesort.BubbleSort;
9:          specialinvoke $r1.<testbubblesort.BubbleSort: void <init>()>();
10:         r2 = $r1;
11:         $r4 = newarray (int)[7];
12:         $r4[0] = 64;
13:         $r4[1] = 34;
14:         $r4[2] = 25;
15:         $r4[3] = 12;
16:         $r4[4] = 22;
17:         $r4[5] = 11;
18:         $r4[6] = 90;
19:         r3 = $r4;
20:         virtualinvoke r2.<testbubblesort.BubbleSort: void bubbleSort(int[])>(r3);
21:         $r5 = <java.lang.System: java.io.PrintStream out>;
22:         virtualinvoke $r5.<java.io.PrintStream: void println(java.lang.String)>("Sorted array");
23:         virtualinvoke r2.<testbubblesort.BubbleSort: void printArray(int[])>(r3);
24:         return;
25:  }
```

Figure 12

The 3-address code form of the security-sensitive method at line 21 in Fig. 10.

```
1.  public class QuickSort {
2.       public static void main(String[] args) {
3.            int[] x = { 9, 2, 4, 7, 3, 6, 10, 5 };
4.            System.out.println(Arrays.toString(x));
5.            int low = 0;
6.            int high = x.length - 1;
7.            quickSort(x, low, high);
8.            System.out.println(Arrays.toString(x));
9.       }
10. public static void QuickSort(int[] arr, int low, int high) {
11.           if (arr == null || arr.length == 0)
12.                return;
13.           if (low >= high)
14.                return;
15.           int middle = low + (high - low) / 2;
16.           @int pivot = arr[middle];
17.           int i = low, j = high;
18.           while (i <= j) {
19.                while (arr[i] < pivot) {
20.                     i++;
21.                }
22.                while (arr[j] > pivot) {
23.                     j--;
24.                }
25.                if (i <= j) {
26.                     int temp = arr[i];
27.                     arr[i] = arr[j];
28.                     arr[j] = temp;
29.                     i++;
30.                     j--;
31.                }}
32.           if (low < j)
33.                quickSort(arr, low, j);
34.           if (high > i)
35.                quickSort(arr, i, high);
36.      }
37. }
```

Figure 13

The Original QuickSort Application.

```
1.  public class QuickSort {
2.       public static void main(String[] args) {
3.            int[] x = { 9, 2, 4, 7, 3, 6, 10, 5 };
4.            System.out.println(Arrays.toString(x));
5.            int low = 0;
6.            int high = x.length - 1;
7.            quickSort(x, low, high);
8.            System.out.println(Arrays.toString(x));
9.       }
10.      public static void QuickSort(int[] arr, int low, int high) {
11.           if (arr == null || arr.length == 0)
12.                return;
13.           if (low >= high)
14.                return;
15.           int middle = low + (high - low) / 2;
16.           @int pivot = arr[middle];
17.           int i = low, j = high;
18.           while (i <= j) {
19.                while (arr[i] < pivot) {
20.                     i++;
21.                }
22.                while (arr[j] > pivot) {
23.                     j--;
24.                }
25.                if (i <= j) {
26.                     int temp = arr[i];
27.                     arr[i] = arr[j];
28.                     arr[j] = temp;
29.                     i++;
30.                     j--;
31.                }
32.           }
33.           if (low < j)
34.                quickSort(arr, low, j);
35.           if (high > i)
36.                quickSort(arr, i, high);
37.      }
38. }
```

Figure 14

The Partitioned QuickSort Application

```
1:      public static void quickSort(int[], int, int){
2:      int[] r0;
3:      int i0, i1, i2, i3, i4, i5, i6, $i7, $i8, $i9, $i10, $i11, $i12;
4:      r0 := @parameter0: int[];
5:      i0 := @parameter1: int;
6:      i1 := @parameter2: int;
7:      if r0 == null goto label0;
8:      $i7 = lengthof r0;
9:      if $i7 != 0 goto label1;
10:     label0:
11:     return;
12:     label1:
13:     if i0 < i1 goto label2;
14:     return;
15:     label2:
16:     $i8 = i1 - i0;
17:     $i9 = $i8 / 2;
18:     i2 = i0 + $i9;
19:     i3 = r0[i2];
20:     i4 = i0;
21:     i5 = i1;
22:     goto label7;
23:     label3:
24:     i4 = i4 + 1;
25:     label4:
26:     $i10 = r0[i4];
27:     if $i10 < i3 goto label3;
28:     goto label6;
29:     label5:
30:     i5 = i5 + -1;
31:     label6:
32:     $i11 = r0[i5];
33:     if $i11 > i3 goto label5;
34:     if i4 > i5 goto label7;
35:     i6 = r0[i4];
36:     $i12 = r0[i5];
37:     r0[i4] = $i12;
38:     r0[i5] = i6;
39:     i4 = i4 + 1;
40:     i5 = i5 + -1;
41:     label7:
42:     if i4 <= i5 goto label4;
43:     if i0 >= i5 goto label8;
44:     staticinvoke <QuickSort: void quickSort(int[],int,int)>(r0, i0, i5);
45:     label8:
46:     if i1 <= i4 goto label9;
47:     staticinvoke <QuickSort: void quickSort(int[],int,int)>(r0, i4, i1);
48:     label9:
49:     return;
50:     }
```

Figure 15

The 3-address code of the sensitive method at line 10 in Fig. 14

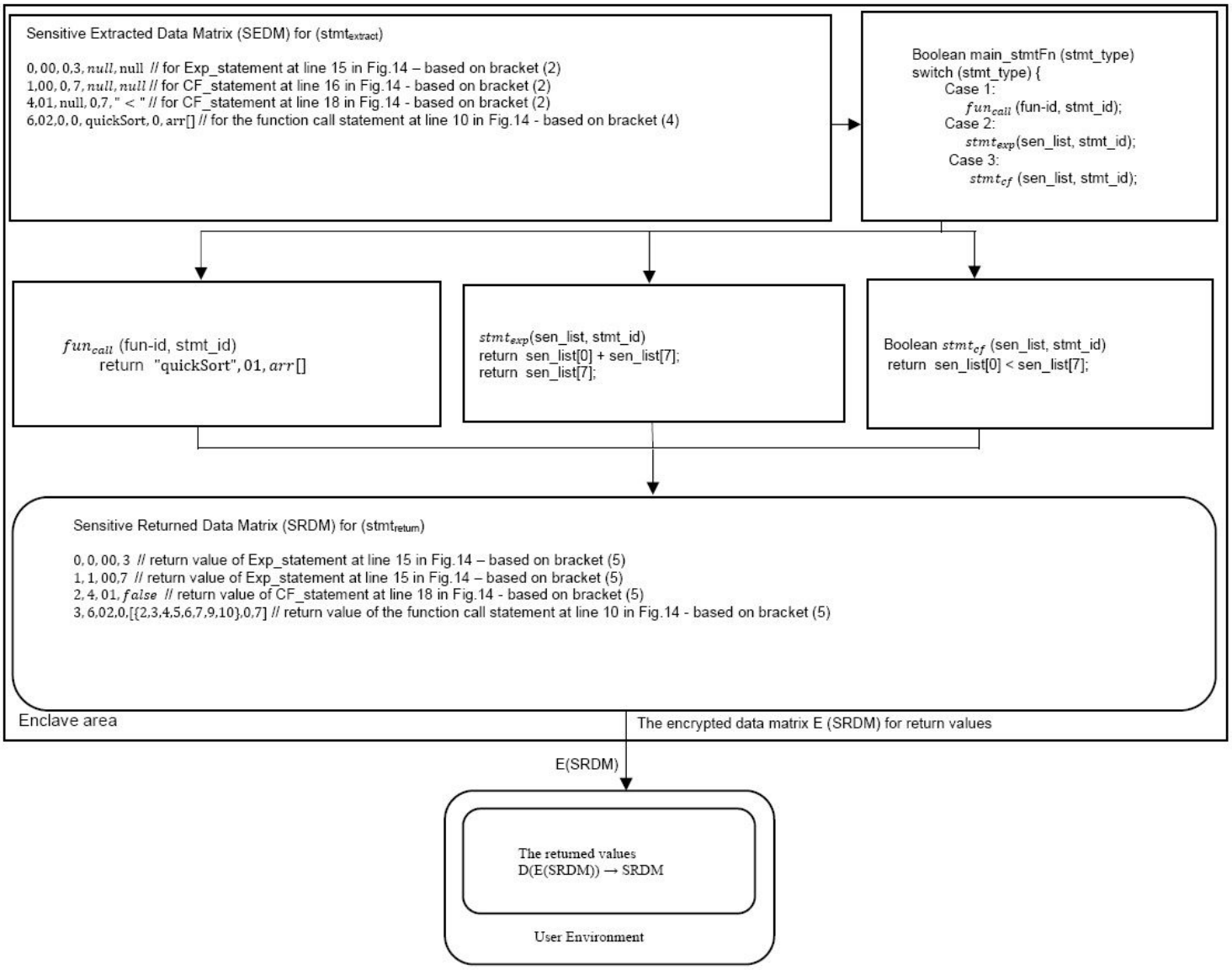

Figure 16

Execution process of QuickSort application inside/outside the enclave

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- TheConferenceVersion.pdf