Only Pay for What You Need: Detecting and Removing Unnecessary TEE-Based Code

Yin Liu[a,*], Siddharth Dhar[b], Eli Tilevich[b]

[a]*Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China*
[b]*Software Innovations Lab, Virginia Tech, 2202 Kraft Drive, Blacksburg, VA 24060, USA*

Abstract

A Trusted Execution Environment (TEE) provides an isolated hardware environment for sensitive code and data to protect a system's integrity and confidentiality. As we discovered, programmers tend to overuse TEE protection. When they place non-sensitive code in TEE, the trusted computing base (TCB) grows unnecessarily, leading to long execution latencies and large attack surfaces. To address this problem, we first study a representative sample of open-source projects to uncover how TEE is utilized in real-world software. To facilitate the process of removing non-sensitive code from TEE, we introduce *TEE Insourcing*, a new type of software refactoring that identifies and removes the *unnecessary* program parts out of TEE. We implemented *TEE Insourcing* as the TEE-DRUP framework, which operates in three phases: (1) a variable sensitivity analysis designates each variable as sensitive or non-sensitive; (2) a TEE-aware taint analysis identifies non-sensitive TEE-based functions; (3) a fully-declarative program transformation automatically moves these functions out of TEE. Our evaluation demonstrates that our approach is correct, effective, and usable. By deploying TEE-DRUP to discover and remove the unnecessary TEE code, programmers can both reduce the TCB's size and improve system performance[1].

Keywords: TEE, Program Analysis, and Code Transformation

1. Introduction

A soaring number of computing devices continuously collect massive amounts of data (e.g., biometric ids, geolocations, and images), much of which is sensitive [2]. Sensitive data and code processing them are the target of many data disclosure and code tampering attacks [3, 4, 5, 6, 7, 8]. An increasingly popular

[*] Yin Liu is the corresponding author.

Email addresses: `yinliu@bjut.edu.cn` (Yin Liu), `siddharth@vt.edu` (Siddharth Dhar), `tilevich@cs.vt.edu` (Eli Tilevich)

[1] This article is a revised and extended version of our prior paper, published in the 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2020) [1]

protection mechanism isolates sensitive code and data from the outside world[2] in a trusted execution environment (TEE) (e.g., SGX [9] and OP-TEE [10]). However, if developers increase volumes of code run in TEE (*not all of that code is sensitive*), then the trusted computing base (TCB) would grow unnecessarily, causing *performance* and *security* issues.

When it comes to performance, prior research identifies the communication between TEE and the outside world as a performance bottleneck that can consume the majority of execution time [11]. For example, numerous function invocations, entering or leaving TEE, trigger a large volume of in/out communication, slowing down the entire system [12]. When it comes to security, prior works [13, 14, 15, 16, 17, 11] move programmer-specified data or functions, even the entire system (e.g., Graphene [18], Haven [19], and SCONE [20]) to TEE. As the trusted computing base (TCB) grows larger, so does the resulting attack surface. Since security vulnerabilities increase proportionally to the code size [21], any vulnerable or malicious functions inside TEE can compromise the security of the entire system. For example, memory corruption attacks can exploit vulnerabilities within a TEE-based function[3] [22, 23]. *Unfortunately, the research community has given much less attention to the status quo of the presence of unnecessary (non-sensitive) code and the resulting TCB sizes in real-world TEE usages.* To bridge this gap, this article first conducts a comprehensively study of more than 400 open-source projects that use TEE. We found substantial evidence of misplacing TEE-based functions, with 61% of the studied projects putting functions into TEE unnecessarily; and the TCB size growing out of control, with 28% of the studied projects putting more than 50% of their total codebase into TEE.

Motivated by our findings, we introduce a new software refactoring —*TEE Insourcing*— that inverses the process of "execution offloading" to reduce the TCB size of legacy TEE projects. *TEE Insourcing* (1) identifies sensitive data; (2) detects TEE-based code not operating on sensitive variables, and as such not needing TEE protection; (3) moves that code to the outside world. Each of these phases presents challenges that must be addressed. Moving sensitive code and data to the outside world would compromise security, so all moving targets must be identified reliably in terms of accuracy, precision, and recall. Sensitive data can flow cross the boundary between TEE and its outside world. However, existing flow analysis techniques are inapplicable to such heterogeneous information flows. To correctly move code out of TEE, a developer must be familiar with both the TEE programming conventions and the program logic, a significant burden to accomplish by hand. However, existing automated program transformation techniques cannot alleviate this burden.

[2] The *normal* (or *outside*) and *secure* worlds are standard TEE terms. In the secure world, code is protected; in the normal world, code is unprotected and compromisable (see section 2).

[3] A TEE-based function is deployed and executed in TEE.

We present TEE-DRUP, the first semi-automated *TEE Insourcing* framework, whose novel program analysis and transformation techniques help infer sensitive code to isolate in TEE, discover the misplaced (non-sensitive) code that should *not* be in TEE, and automatically move the discovered non-sensitive code to the outside world. In phase (1) above, an NLP-based variable sensitivity analysis designates program variables as sensitive or non-sensitive, based on their textual information. Guided by these designations, developers then verify and confirm which variables are sensitive. In phase (2), a novel TEE-aware taint analysis links each TEE-based function to its *normal world counterparts*[4] to identify those TEE-based functions that never operate on developer-confirmed sensitive variables. These functions are then flagged as non-sensitive, so developers can verify and confirm which ones of them are to move to the outside world. In phase (3), exposed via a declarative meta-programming model, an automated transformation modifies the system's intermediate representation (IR) to move the developer-confirmed non-sensitive functions to the outside world.

Based on our evaluation, TEE-DRUP distinguishes between sensitive and non-sensitive variables with satisfying accuracy, precision, and recall (the actual values are greater than 80% in the majority of evaluation scenarios). Further, moving non-sensitive code out of the TEE always improves the overall system's performance (the speedup factor ranges between 1.35 and 10K). Finally, TEE-DRUP's automated program analysis and transformation require only a small programming effort.

The contribution of this article is as follows:

1. *A comprehensive study of real-world TEE practices.* To our best knowledge, we conducted the first study of over 400 open-sourced TEE-related projects to investigate the status quo of using TEE in real-world software development.
2. ***TEE Insourcing***, a novel approach to reducing the TCB size, concretely realized as TEE-DRUP that offers:
    - *a variable sensitivity analysis* that by using NLP designates program variables as sensitive or non-sensitive, so developers can verify and confirm sensitive variables.
    - *a TEE-aware taint analysis* that flags TEE-based functions not operating variables, so developers can verify and confirm non-sensitive functions.
    - *a compiler-assisted automated program transformation, supported by a declarative meta-programming model* that moves TEE-based non-sensitive functions to the outside world to satisfy dissimilar requirements.

[4] "A normal world counterpart" is a function in the normal (or outside) world that directly invokes (or is invoked by) a TEE-based function.

3. **An empirical evaluation** of TEE-DRUP's (a) correctness in distinguishing between sensitive and non-sensitive variables, (b) effectiveness in improving system performance, and (c) low programming effort.

The rest of this article is organized as follows: Section 2 clarifies the definitions, enabling technologies, and assumptions in this research; Section 3 describes our study of real-world TEE practices; Section 4 gives an overview of TEE-DRUP with an example; Section 5, 6, and 7 detail the entire process of identifying and migrating unnecessary code in TEE. Section 8 presents and discusses our evaluation results. Section 9 compares TEE-DRUP to the related state of the art. Section 10 presents concluding remarks.

2. Definitions, Enabling Technologies, and Assumptions

We use the following definitions, techniques, and assumptions for our approach:

2.1. Definitions

Sensitive & non-Sensitive: "Sensitive" describes all security-related objects (e.g., passwords, decryption keys, memory addresses) and operations (e.g., access control, encryption, memory accessing), with the rest considered "non-sensitive". These objects and operations correspond to what SANS[5] refers to as "security terms" [24]. *Sensitive* data (or variables) store security-related information or are referenced in security-related operations. *Sensitive* code (or functions) operates on *sensitive* data.

Normal & Secure worlds: The normal and secure worlds are standard TEE terms. In the secure world, code is protected, while in the normal world unprotected. We also use the term "the outside world" to refer to "the normal world", and "TEE" to refer to "the secure world."

2.2. Enabling Technologies

Intel's Software Guard Extensions (SGX) [9]: To protect the integrity and confidentiality of sensitive code and data, Intel's SGX isolates a protected memory region—**enclave**—for trusted execution. Hence, for a system to use SGX, its code must be divided into trusted and untrusted parts, with the former running inside *enclaves* and the latter outside. To preserve the *enclave*'s trusted execution, the untrusted part can invoke enclave functions (i.e., *ECalls*) only via SGX-provided communication channels. TEE-DRUP's design and implementation focuses on SGX, due to the maturity of this TEE implementation.

Natural language processing (NLP): NLP techniques for processing natural languages have been used for identifying security information in program code [25, 26, 27]. TEE-DRUP's NLP-based sensitivity analysis designates sensitive variables that should be protected in TEE.

[5] An authoritative source for information security training, certification, and research.

Taint Analysis is a powerful program analysis technique that tracks how the tainted data flows during program execution. Mobile and IoT security researchers have applied taint analysis to detect malicious behaviors and code vulnerabilities [28, 29, 30, 26, 31]. TEE-DRUP's taint analysis confirms whether sensitive variables are accessed by TEE-based functions.

LLVM *[32]* is a mature compiler infrastructure used for program analysis at the source-code and binary levels. For source-code analysis, LLVM features `libtooling tool` [33], while for binary analysis, it features `Pass` [34]. TEE-DRUP customizes `libtooling tool` to extract the variable information from a system's source code, and introduces a series of new `Pass`es that removes non-sensitive functions from TEE to run outside.

2.3. Assumptions and Scope

We assume *1) the attacks are carried out only at runtime.* That is, attackers would not be able to modify the source code to mislead TEE-DRUP's sensitivity analysis and taint analysis; *2) developers name variables, functions, and files in a meaningful way that actually describes their usage or purpose.* For example, it is highly probable that a variable named "password" would represent some password-related information. In fact, major IT companies, including Google [35], IBM [36], and Microsoft [37], have established coding conventions requiring that program identifiers be named intuitively. Regular code reviews often come up with suggestions how to rename identifiers to more meaningfully reflect their roles and usage scenarios [38]; and *3) the users of TEE insourcing are application programmers who may not be knowledgeable enough to be able to reliably identify sensitive data and code.* This assumption is realistic, as one cannot expect an application programmer to possess expertise in security and TEE as well as the system's business logic. Even the primary contributors to a major software project commonly forget the project's fine-grained implementation details [39]. *In addition, since major TEE implementations (e.g., SGX and OP-TEE) only work with the C language, we only support C projects.*

3. Analyzing Real-world TEE Practices

We analyzed a large set of GitHub repositories to identify how TEE is used in open-source software projects. Our goal is to answer the following research questions:

- **RQ1 (Unnecessary TEE-based code):** *Which percentage of projects uses TEE effectively?*
- **RQ2 (TCB size):** *What is the average TCB size in the studied projects?*
- **RQ3 (Features in TEE):** *Which features are placed into TEE most frequently?*

We first introduce our data collection procedure, and then discuss the research questions above in turn.

3.1. Data Collection

To obtain a realistic snapshot of the common usage of TEE[6] in open-source software, we used GitHub as one of the most popular and largest repositories. To retrieve those repositories whose source code integrates Intel SGX, we used GitHub's search API [40]. Specifically, we searched for the repositories in which any source file includes the *sgx_urts.h* header. Before running any secure code, a SGX based project must first create an enclave[7] by invoking the *create_enclave()* function, whose header is in *sgx_urts.h.* To automate the process, we created a Python script that automatically fetches the repositories that match the aforementioned search criteria. The script creates a list of URLs of the matched GitHub repositories to collect a total of 429 projects.

3.2. RQ1: Unnecessary TEE-based code

As recommended by Intel [41] and other guidelines [42, 43], we categorized the projects to identify if they use TEE effectively. The parameters used in this identification process are based on the set of features that were put inside the enclaves.

We manually analyzed each repository to identify the features that incorporate SGX enclaves. The analysis places its subjects into one of the following categories:

1. *Not Relevant:* a repository contains no sensitive features in the enclaves, suggesting that these features can be moved out.
2. *Needs Refactoring:* a repository contains an enclave whose content contains code that could have been placed in the outside world.
3. *Relevant:* a repository's enclaves contain the code of only sensitive features.
4. *N/A:* a repository does not use TEE but is a wrapper for a TEE or a library that makes use of TEE, or a benchmark tool. However, the project itself is not a TEE-based.

Table 1: Repository Categorization

Category	**Number of Repositories**
Relevant	133
Not Relevant	129
Needs Refactoring	82
N/A	85
Total	429

[6] In this paper, we focus on Intel SGX

[7] Enclave is the secure execution environment for an Intel SGX based system

Results. Table 1 presents high-level results of the analysis of the GitHub repositories.

Recall that we pre-selected those repositories in which at least one source file included *sgx_urts.h*. Out of the resulting 429 repositories, 85 repositories had no code running in TEE, while the remaining 344 repositories had some code isolated in TEE. These remaining repositories were selected for further analysis.

Out of the total 344 repositories, only 133 of them were found to use TEE effectively, isolating only sensitive features, related to security or privacy. These projects were labelled as 'Relevant'. The remaining 211 repositories (i.e., 61%) were found as either 'Not Relevant' or 'Needs Refactoring'.

Answer to RQ1: 61% of the GitHub repositories that use Intel SGX, either do not use the enclave for any sensitive features or include code in the enclave that does not cater to any sensitive code mixed with the features that actually need to be isolated in the enclave.

3.3. RQ2: TCB size

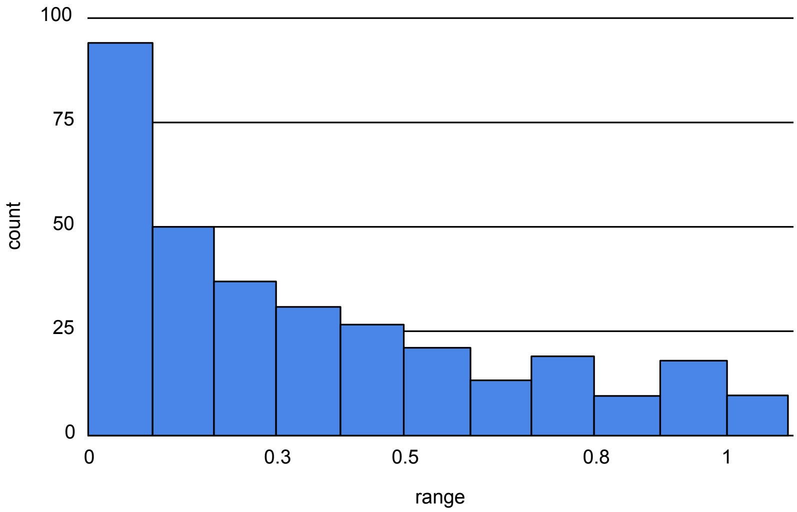
In addition to accurately identifying features that should be protected in the TEE, it is also important to ensure that the ratio of the resulting trusted computing base (TCB) to the complete project is as small as possible. As TEE hardware is known to be slower than the main CPUs[8], large TCBs can incur a high performance overhead. More importantly, invoking each TEE-based functionality is expensive, due to the significant communication overhead between the main operating system and the TEE.

We used the Python library *pygount* [44] to calculate the TCB ratio for the analyzed projects. Given a project, we first use *pygount* to get its total number of lines of code (LoC). Then, we found all the Enclave Definition Language (edl) files which have a *.edl* extension in the project and stored the directory in which the *edl* file is placed. These *edl* files are the interface to the enclave, and all enclave code is most likely placed in the same directory as the *edl* file. Once we had the directories of the *edl* files, we used *pygount* again to count the lines of code within these directories to get the lines of code in the enclave. The TCB size was then calculated as:

$$TCB\ Ratio = \frac{Enclave\ lines\ of\ code}{Total\ lines\ of\ code} \tag{1}$$

Results. Among the total of 429 projects, we obtained 333 valid values of the TCB ratio that is in the range from 0 to 1, with the average of 0.341 and the median of 0.251. Furthermore, as shown in Figure 1, among the 333 valid projects, 93 (28%) projects' TCB ratio is greater than 0.5. That is, in these 93 projects, their code inside TEE accounts for more than 50% of their total lines of code.

[8] One reason is that TEE hardware usually has limited resources (e.g., memory).



x-axis: the values of TCB ratio ranging from 0 to 1.

y-axis: the number of projects whose TCB ratio falls in each value scale (i.e., 0.1).

Figure 1: TCB Ratio

Answer to RQ2: among the 333 projects, for which we obtained their valid TCB ratio, about one third of them put more than a half of their entire codebase into TEE.

3.4. RQ3: Features in TEE

To understand which program features are typically isolated, we further analyzed the 133 "Relevant" 175 repositories. Specifically, we manually examined their enclave code, extracting TEE-isolated features, having identified 65 such features in total.

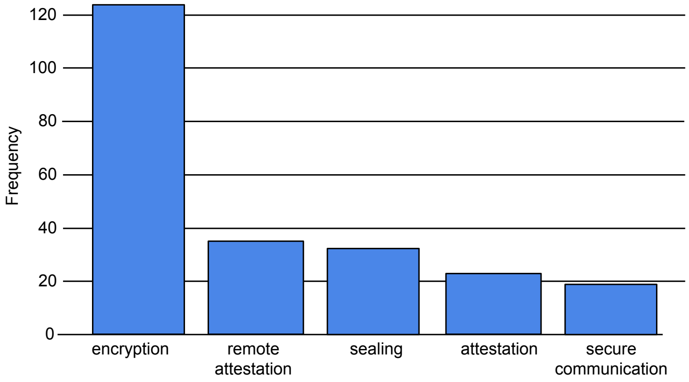


Figure 2: Top 5 TEE features as based on their frequency

Results. Fig 2 depicts the top 5 most frequent features along with their frequencies. As per this figure, the feature ‘encryption’ has by far the highest frequency. The other 4 features ‘remote attestation’, ‘sealing’, ‘attestation’, and ‘secure communication’ have approximately similar frequencies. The remaining 60 features occurred only infrequently (i.e., between 2 and 10 times across all repositories).

Answer to RQ3: among the features effectively placed in TEE, all of the top 5 most frequent features contain security-related operations.

3.5. Discussion

Based on our findings, we conclude that (1) the function misplacing issue is real: developers do overuse TEE by isolating code and data that are not sensitive; (2) the TCB size does grow out of control: a considerable percentage of projects put more than a half of their entire codebase into TEE; (3) since the most frequent features do contain security-related operations, one can reliably identify whether a function should be placed into TEE by checking if it contains security-sensitive operations.

4. Solution Overview

Guided by above findings, we created TEE-DRUP to assist developers in identifying sensitive code to be protected in TEE and also reducing the TCB size by migrating non-sensitive code out of TEE. We first describe the process of applying TEE-DRUP, and then present an example application.

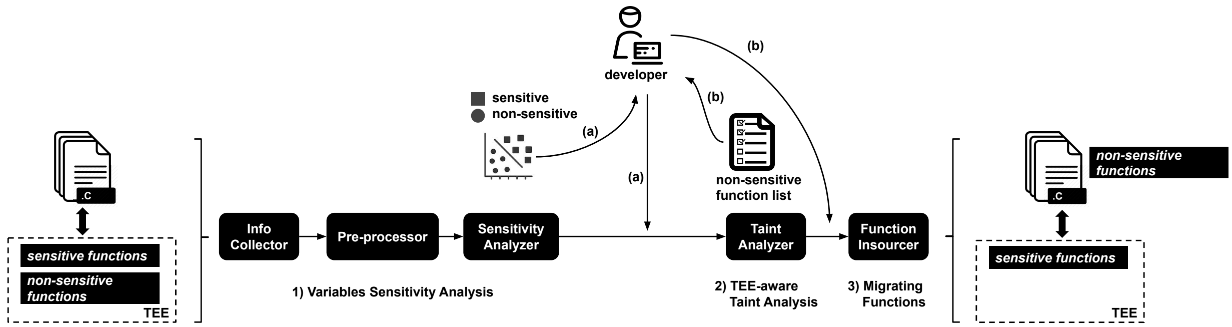


Figure 3: TEE-DRUP overview

4.1. TEE-DRUP *Process*

Figure 3 shows TEE-DRUP’s three main phase: (1) analyzing the sensitivity of variables (i.e., `Info Collector`, `Pre-processor`, and `Sensitivity Analyzer`), (2) identifying non-sensitive functions in TEE (i.e., `Taint Analyzer`), and (3) migrating the non-sensitive functions out of TEE (i.e., `Function Insourcer`).

Phase (1) first applies `Info Collector` to obtain each variable's textual descriptions (i.e., variable name, type name, function, and file path). Then, it encodes the collected information as a textual vector for further analysis. Then Phase (1) applies `Pre-processor` to merge duplicated vectors and remove unnecessary information. Finally, Phase (1) applies `Sensitivity Analyzer` that by means of NLP computes each variable's sensitivity level from its textual attributes (i.e., name, type, file path, etc.) and designates sensitive variables. With the designated variables at their disposal, developers then verify and confirm which variables are indeed sensitive (i.e., step (a) in Figure 3).

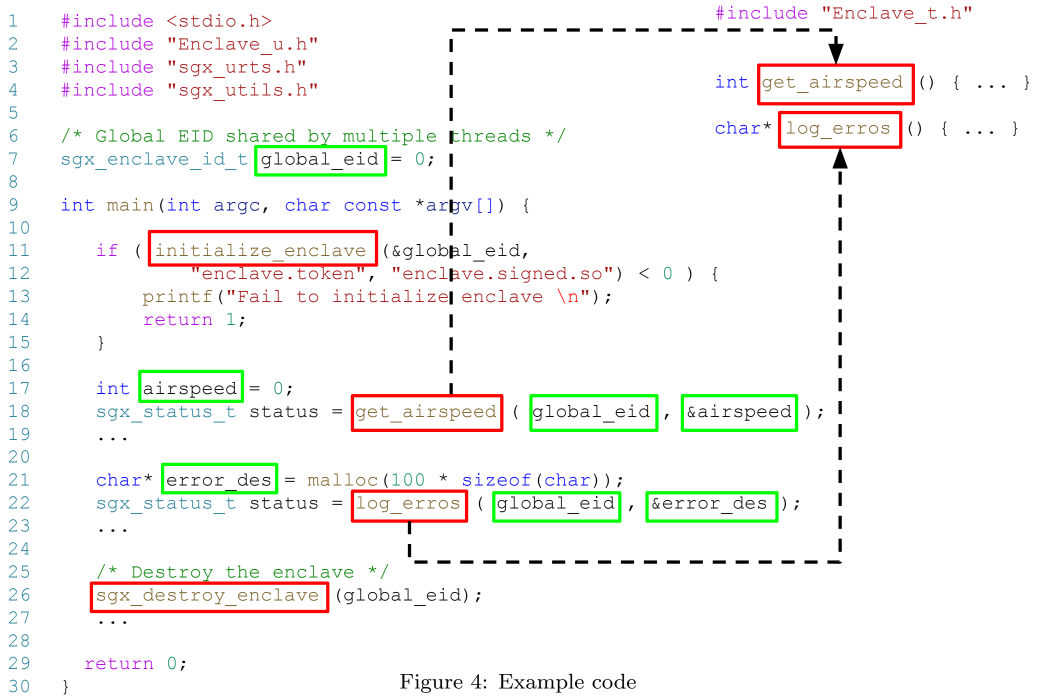
Phase (2) applies `Taint Analyzer`, which takes as input developer-specified sensitive variables and outputs which TEE-based functions are non-sensitive. Its dataflow-based traversal detects those TEE-based functions that never operate on sensitive variables. With the reported non-sensitive functions at their disposal, developers then verify and confirm which functions are to be moved to the outside world (i.e., step (b) in Figure 3).

Phase (3) applies `Function Insourcer` to automatically adjust the TEE-related call interfaces, remove their TEE metadata[9] from TEE, and merge developer-confirmed non-sensitive functions with those in the code outside of TEE. It is through these steps that TEE-DRUP keeps the sensitive functions in TEE, while moving the non-sensitive functions out.

4.2. A Motivating Example

Consider the following example that demonstrates how TEE-DRUP analyzes and modifies the code of a legacy system that uses TEE. This example's code is adapted from standard official samples of SGX-based code [45], with the side-by-side code snippets appearing in Figure 4. On the left, function `main` is in the outside world, while in the right corner, there are two TEE-based functions (i.e., `get_airspeed` and `log_erros`) invoked from `main`. Within `main`, `get_airspeed` and `log_errors` invoke the corresponding TEE-based functions with the same name, an example of "normal world counterparts" of TEE-based functions. The SGX terminology refers to trusted execution regions as "enclaves" and TEE-based functions as "ECalls." An enclave is identified by its "enclave ID" (i.e., `global_eid` on line 7). To invoke ECalls, an enclave ID should be passed as the first parameter (lines 18 and 22). Further, to obtain ECalls' return value, a pointer should be provided as another extra parameter (lines 17 and 21). Moreover, initialization (`initialize_enclave`, line 11) and cleanup (`sgx_destroy_enclave`, line 26) functions must be called both before and after interacting with a TEE-based function.

[9] Metadata is used by TEE-related call interfaces only (e.g., enclave's IDs in SGX).



```
1    #include <stdio.h>
2    #include "Enclave_u.h"
3    #include "sgx_urts.h"
4    #include "sgx_utils.h"
5
6    /* Global EID shared by multiple threads */
7    sgx_enclave_id_t global_eid = 0;
8
9    int main(int argc, char const *argv[]) {
10
11      if ( initialize_enclave (&global_eid,
12              "enclave.token", "enclave.signed.so") < 0 ) {
13          printf("Fail to initialize enclave \n");
14          return 1;
15      }
16
17      int airspeed = 0;
18      sgx_status_t status = get_airspeed ( global_eid , &airspeed );
19      ...
20
21      char* error_des = malloc(100 * sizeof(char));
22      sgx_status_t status = log_erros ( global_eid , &error_des );
23      ...
24
25      /* Destroy the enclave */
26      sgx_destroy_enclave (global_eid);
27      ...
28
29     return 0;
30   }
```

```
#include "Enclave_t.h"

int get_airspeed () { ... }

char* log_erros () { ... }
```

Figure 4: Example code

The aforementioned three phases work as follows. For each variable (i.e.,`airspeed`, `error_des`, and `global_eid`), `Info Collector` extracts and encodes their textual attributes into corresponding attribute vectors. Then, `Pre-processor` removes the `global_eid`'s vector, as it only identifies a SGX enclave. After that, `Sensitivity Analyzer` marks the sensitivity of `airspeed` as 6[10], and that of `error_des` as 3. Based on these sensitivity level, `Sensitivity Analyzer` designates `airspeed` as sensitive variable while `error_des` as non-sensitive variable. Having examined the designations, the developer verifies and confirms `airspeed` as the sensitive variable. Using the sensitive variable (i.e., `airspeed`) as the source, and TEE-based functions (i.e., `get_airspeed` and `log_erros`) as the sink, `Taint Analyzer` discovers that only `get_airspeed` manipulates the sensitive variable (i.e., `airspeed` on line 18). Thus, `Taint Analyzer` generates a function list, in which `log_erros` is marked as "non-sensitive." The developer then verifies and confirms `log_erros` to be moved to the outside world. Finally, `Function Insourcer` extracts `log_erros` into a program unit executing outside the enclave, and redirects its callers to invoke the extracted code instead[11].

[10] Here "6" and the following numbers in this Section are the example value (not real) for demonstrating our solution only.

[11] If all functions are moved outside the TEE, `Function Insourcer` will remove the unnecessary metadata and functions (i.e., `global_eid`, `initialize_enclave` and `sgx_destroy_enclave`) to the outside world.

4.3. Assumptions and Scope

We assume *1) attackers would not be able to modify the source code to mislead* TEE-DRUP *'s sensitivity and taint analyses*; and *2) developers name variables, functions, and files descriptively.* For example, it is highly probable that a variable named "password" would represent some password-related information. In fact, major IT companies, including Google [35], IBM [36], and Microsoft [37], have established coding conventions requiring that program identifiers be named intuitively. Regular code reviews often come up with suggestions how to rename identifiers to more meaningfully reflect their roles and usage scenarios [38]. *In addition, since major TEE implementations (e.g., SGX and OP-TEE) only work with the C language, we only support C projects.* We plan to extend this support to managed and multi language projects as a future work direction.

5. Analyzing Variables Sensitivity

To help developers identify which variables are sensitive, TEE-DRUP offers a variables sensitivity analysis that determines how and which textual information of a variable to collect (5.1, 5.2); how to determine a variable's sensitivity level from the collected textual information (5.3); and how to compute a threshold to designate variables as (non-) sensitive (5.4).

5.1. Collecting Information

1) Extracting Program Data. Given a parsed representation of a system's source code, `Info Collector` traverses the abstract syntax tree (AST) to locate variable nodes and collects their textual information. The collected information for each variable includes the variable's name, its type's name, its enclosing function's name, and the source file's path. This textual information determines the sensitivity level of each program variable. `Info Collector` is implemented as a LLVM *libtooling tool*.

2) Encoding Variable Data. `Info Collector` encodes the extracted variables' information into a format that facilitates the subsequent operations for analyzing sensitivity. To that end, each variable is associated with the *textual-info* records, which stores the variable's textual description (encoded into string vectors). Table 2 shows an encoded record for variable `const int * the_password`.

Table 2: Data Format: `const int * the_password;`

id	var. name	type	function	filepath
7	the_password	int	config	src/wifi_config.h

5.2. Pre-processing

1) Filtering Records. Because realistic systems can use a large number of variables, calculating sensitivity levels for each of them could be quite computationally intensive. However, not all variables need to be analyzed. To reduce the amount of required computation, `Pre-processor` identifies and removes the unnecessary variable records. These variables are those that used exclusively within SGX enclaves and those with names so short that the variable's usage cannot be discerned. For example, enclave IDs are used only within SGX enclaves, while variables named `i` or `j` are not indicative of their usage. In addition, `Pre-processor` merges duplicated records.

2) Splitting Identifiers. Even for meaningfully named variables, types, functions, and file path, their names can follow dissimilar naming conventions, such as delimiter-separated (e.g., the_password) or letter case-separated (e.g., thePassword). To be able to process these names irrespective of their naming conventions, the identifiers containing convention-specific characters are split into separate parts (e.g., thePassword and the_password would become identical arrays of "the" and "password.")

3) Removing Redundancies. some parts of identifiers are indicative of their construct's sensitivity, but others carry no such information. For example, in "the_password", "password" indicates high sensitivity, while "the" provides no additional sensitivity information. Hence, `Pre-processor` performs dictionary-based removal of identifier parts that correspond to prepositions (e.g., in, on, at), pronouns, articles, and tense construction verbs (i.e., be, have, and their variants). In our example, "password" will be retained, but "the" will be removed.

5.3. Computing Sensitivity Levels

Although it is difficult to reliably determine and quantify a variable's sensitivity, `Sensitivity Analyzer` offers an NLP-based algorithm that provides a reasonable approximation. In short, TEE-DRUP computes the similarity between a word in question and the words in the dictionary of security terms. The similarity then determines the word's most likely sensitivity level.

1) Rationales. When computing the similarity, the variable's name, its type, function and file path are taken into account as guided by the following rationales [46]:

a) Sensitive variables tend to appear in certain functions and files. For example, the variable "the_password" is more likely to be sensitive if it is referenced by the "login" function in the "login.c" file rather than the "unit_test" function in the "test_cases.c" file.

b) If a variable's type is developer-defined (e.g., struct/class/union names), the resulting type name can be indicative of its sensitivity. For example, as our evaluation demonstrates (8), many such variables in our evaluation subjects (e.g., `struct passwd` in project "su-exec") have type names that reveal their variables' sensitivity.

c) Semantic-connections are closer for adjacent rather than nonadjacent words. That is, if an identifier appears alongside a known sensitive identifier, it is likely to be semantically related to sensitive information. For example, the variable "key" in the path "mapping/a/b/encryption/xx.c" should be more sensitive than in the path "encryption/a/b/mapping/xx.c", because "key" is closer to "encryption" in the former case. That is, the variable "key" is more likely to store the key of encryption rather than the key of key-value pairs for mapping.

d) A variable's textual information impacts its sensitivity level to varying degrees. Our variable sensitivity analysis involves four kinds of textual information: variable name, type name, function name, and file path, each of which impacts its variable's sensitivity level dissimilarly. The variable name has the highest impact, followed by type and function names, with file path the lowest. When computing a variable's sensitivity level, each of its textual information components is weighted accordingly.

Algorithm 1: TEE-DRUP's variable labeling.

```
Input  : textual_info_list (i.e., variables' textual info list)
         dict (i.e., a collection of security terms)
         $\lambda$ (i.e., the attenuation rate for file paths)
Output: variables with sensitivity levels

Function: get_similarity(word_array, $dict$, $\lambda$):
$sim \leftarrow 0$
foreach $word : word\_array$ do
    $txt \leftarrow find\_most\_similar(word, dict)$
    $d \leftarrow similarity(word, txt)$
    increase $sim$ by $d * \lambda$
end
$avg \leftarrow average(\mathrm{sim})$
return $avg$

Function: calculating_main(textual_info_list, $dict$):
foreach $var : textual\_info\_list$ do
    /* for variable name.                                    */
    $var\_name \leftarrow get\_var\_name(var)$
    $sim\_var \leftarrow get\_similarity(\mathrm{var\_name,dict}, 1)$
    /* for type name.                                        */
    $type\_name \leftarrow get\_type\_name(var)$
    $sim\_type \leftarrow get\_similarity(\mathrm{type\_name,dict}, 1)$
    /* for function name.                                    */
    $func\_name \leftarrow get\_func\_name(var)$
    $sim\_func \leftarrow get\_similarity(\mathrm{func\_name,dict}, 1)$
    /* for file path.                                        */
    $path \leftarrow get\_path(var)$
    $sim\_path \leftarrow get\_similarity(\mathrm{path,dict}, 0.8)$
    $var.sensitivity \leftarrow (\mathrm{sim\_var} + \mathrm{sim\_func} * 0.8 + \mathrm{sim\_type} * 0.8 + \mathrm{sim\_path} * 0.5)$
end
```

2) Sensitivity Computation Algorithm. Algorithm-1's function `calculating_main` outputs variables with sensitivity levels, given the variables' textual information list (i.e., `textual_info_list`) and a security term dictionary (i.e., `dict`). First, each variable's textual information is obtained (line 11). Then, identifiers are extracted from the pre-processed variable's name, type, function, and file path (lines 12,14,16,18). Next, function `get_similarity` computes the similarity between an extracted identifier and known security terms (lines 13,15,17,19). Each extracted identifier is broken into constituent words (e.g., `error_des` is broken into `error`, `des`). For each word, the algorithm computes the similarity to the most closely similar known security term (lines 4 and 5). The similarities are accumulated (line 6), averaged (line 8), and returned (line 9). An attenuation rate, λ, differentiates adjacent vs. nonadjacent semantic connections. Next, the obtained similarities are weighted as follows: "1"–variable name, "0.8"–type and function names, and "0.5"–file path. These weighted similarities are summed into the variable's sensitivity (line 20).

3) An Example: Following the example in 4.2, consider how `Sensitivity Analyzer` would calculate the sensitivity levels for variables in the input textual-info list. The textual-info list contains variable `airspeed` with type `struct ControlData`, and variable `error_des` with type `struct ReportInfo`. `airspeed` is accessed in function `get_my_airspeed`, defined in "src/navigator.c", while `error_des` is accessed in function `report_for`, defined in "report/log.c".

Pre-processing creates the arrays of `airspeed`'s name, type, function, and file path to [`airspeed`], [`control, data`], [`get, airspeed`], and [`src, navigator`], respectively; while that of `error_des` to [`error, des`], [`report, info`], [`report`], and [`report, log`], respectively.

To demonstrate how sensitivity levels are calculated, take the variable `error_des` as an example: `Sensitivity Analyzer` first obtains the first word `error` from `error_des`'s name array [`error, des`], finds its closest security term (assume the term is "exception") from the dictionary of security terms, and calculates the similarity value (assume the value is 0.8). Afterwards, `Sensitivity Analyzer` obtains the second word `des`'s similarity (assume the value is 0.2). Then, `error_des`'s variable name array [`error, des`]'s similarity is calculated as 0.5 (i.e., $(0.8 + 0.2)/2$). Similarly, `Sensitivity Analyzer` computes the similarities of `error_des`'s type, function, and path arrays. Finally, the computed similarities are weighted and summed into `error_des`'s sensitivity.

Note that, when calculating the similarity for the file path array, `Sensitivity Analyzer` will scale the result by λ (the value is 80% by default). That is, for `error_des`'s file path array [`report, log`], `Sensitivity Analyzer` scales the similarity of `report` by 80%. If the original similarity of `report` is 1, the it will be 0.8 after the scaling (i.e., the original value multiplies λ: $1 * 80\%$).

Similarly, variable `airspeed`'s sensitivity level will also be calculated and assigned. Assume that `airspeed`'s sensitivity level is 3 while that of `error_des` 1.5. Since $3 > 1.5$, the semantic meaning of `airspeed` is closer to known security terms, so it is more sensitive than `error_des`.

4) Implementing Sensitivity Analysis. Our algorithm computes the similarity with *Word2vec*, a Google's word embedding tool [47]. The dictionary of security terms (i.e., security-related objects and operations) comes from SANS, recognized as one of the largest and trusted information security training, certification, and research sources [24]. Further, with TEE-DRUP, developers can add their own words to the dictionary of security terms. For example, a developer can add "airspeed" as a security term, or customize the attenuation rate (i.e., λ), thus potentially improving the accuracy. Despite its heuristic nature and inability to handle certain corner cases, TEE-DRUP's `Sensitivity Analyzer` turned out surprisingly accurate in generating meaningfully sensitivity levels that can guide the developer, as we report in 8.

5.4. Designating Variables as (non-)Sensitive

Given the computed sensitivity levels of the program variables, developers then designate variables as either sensitive or non-sensitive. TEE-DRUP provides an automatic designation algorithm, inspired by the P-tile (short for "Percentile") thresholding method, a classic method that calculates the threshold based on a given percentile [48]. Specifically, given a percentage of program variables, if a variable's sensitivity is higher than the given percentage, TEE-DRUP designates it as sensitive; if lower, non-sensitive. For example, given a percentage "1%", TEE-DRUP would designate the top 1% variables as sensitive, while the bottom 1% as non-sensitive. By default, TEE-DRUP recommends the percentages of 10%, 20%, 30%, 40%, and 50%. Note that, although our evaluation indicates the effectiveness of our designation heuristic, it simply follows empirical principles. Developers can always rely on other mechanisms in search for higher accuracy.

6. Identifying Non-Sensitive Functions

Developers manually confirm which TEE-DRUP-designated variables are indeed sensitive via a custom annotation `sens`. TEE-DRUP then automatically identifies those TEE-based functions that reference none of the annotated variables. Such functions are to be moved out of TEE to the normal world. To that end, we developed a novel *TEE-aware taint analysis* that (1) models call graph and data flow across both the normal and secure worlds of TEE projects (6.1); (2) determines whether a given *TEE-insourcing* can be performed (6.2).

6.1. TEE-Aware Taint Analysis

Since SGX invokes functions from the normal to the secure world (ECall) and vice versa (OCalls), our analysis statically traces information flows both from sensitive sources (variables) in the normal world to trusted sinks (functions) in the secure world and conversely from trusted sources (functions) in the secure world to sensitive sinks (variables) in the normal world. Then, it connects the traced information flows by building an entire call graph across the normal and secure worlds.

1) building TEE-aware Information Flow Graphs. Our analysis first generates individual call graphs for the normal and secure worlds. Then, these call graphs are combined into a TEE-aware Call Graph (TCG) by mapping each TEE-based function to its *normal world counterpart*[12]. To be able to locate *normal world counterparts* efficiently, the analysis uses the *TEE-aware function mapping* data structure, which maps TEE-based function signatures (name, parameters, source file) to the corresponding normal world function signatures. *TEE-aware function mappings* are built by consulting the SGX EDL files. Similarly, by consulting the built TCG, the analysis builds and conflates the data-flow graphs of the normal and secure worlds. Our analysis disregards function pointers, which are unsupported in SGX [49].

2) modeling TEE-aware Taint Flow. To model taint flows across the normal and secure worlds, our analysis treats each annotated variable as both the *source* and the *sink*. As is shown in Figure 5, as a *source*, a variable (i.e., data) can flow to Ecalls (i.e., *sink* in this case) in SGX through their *normal world counterparts* (i.e., ❶❷). Also, as a *sink*, a variable can be impacted by the data flowing from Ecalls (i.e., *source* in this case) and their *normal world counterparts* (i.e., ❸❹). For these two cases (❶❷, and ❸❹), TEE-DRUP performs forward and backward analysis, respectively. Specifically, TEE-DRUP first obtains the system's data-flow graph. Then, with the annotated sensitive variables as the entry point, TEE-DRUP forward traverses the graph to check whether the variables' value flows to the ECalls (i.e., ❶ → ❷), and backward traverses the graph to check whether the variables' value is impacted by the ECalls (i.e., ❸ ← ❹). If the source (i.e., sensitive variables in case ❶❷, ECalls in case ❸❹) flows into the sink (i.e., ECalls in case ❶❷, sensitive variables in case ❸❹), the ECalls are marked as sensitive, while the remaining functions are marked as non-sensitive. All analysis algorithms are implemented as custom LLVM passes.

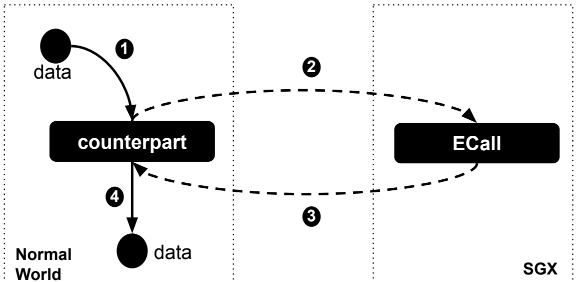


Figure 5: TEE-DRUP's taint-flow analysis

[12] "A normal world counterpart" is a function in the normal world that directly invokes (or is invoked by) a TEE-based function (see an example in 4.2.)

6.2. Insourcing Rules

1) call frequency: A TEE-based non-sensitive function could frequently invoke (or be invoked by) functions in both the secure and normal worlds. Consider a TEE-based function with 10 callers and 10 callees in the secure world, and 5 callers and 5 callees in the normal world. If the invocations from the secure world are more frequent than those from the normal world, it would be beneficial for the function to remain in the secure world, so as to avoid the high inter-world communication costs. To assist developers, our analysis extracts and matches the number callers and callees for each non-sensitive TEE-based function, grouping them into the normal and secure worlds. Further, our loop analysis labels each caller/callee as *in* or *out* of a loop. All the analysis results are presented to developers to help make an informed decision whether to move each TEE-based non-sensitive function out of the TEE.

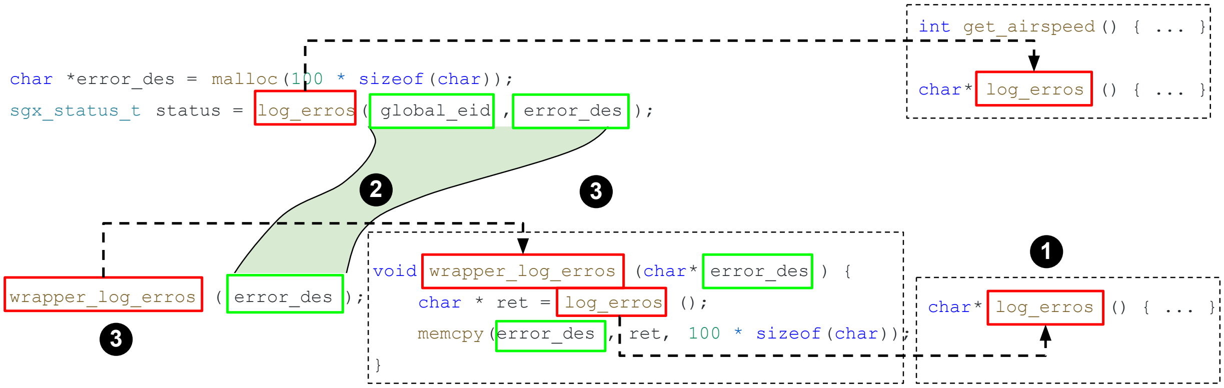
2) global variable: A global variable and functions accessing it must be co-located in the same world. Hence, a non-sensitive TEE-based function accessing a global variable cannot move to the normal world, as distributed global states cannot be maintained. Our analysis detects and reports such occurrences, so developers can decide whether to keep a function in TEE or add logic for maintaining a distributed global state.

7. Insourcing TEE-based Functions

From the suggested list of non-sensitive functions, developers confirm which ones were placed in TEE by mistake (7-I). `Function insourcer` then insources the confirmed non-sensitive functions from the TEE to the outside world (7-II).

I. Refactoring Interface. For developers to specify functions-to-insource, TEE-DRUP provides a custom annotation, `nonsens`, to mark non-sensitive functions. It is through this annotation, `Function Insourcer` identifies which functions to extract and migrate from the secure world to the normal world. The design and implementation of all our custom annotations (including `sens` in 6 and `nonsens`) follow the *Clang annotation scheme* [50] and the GNU style [51].

II. Insourcing Process. `Function insourcer` moves the relevant ECalls outside SGX in the following 3 steps: ❶ **extract ECalls:** by customizing an existing LLVM Pass (`GVExtractionPass`), `Function insourcer` extracts the annotated ECalls from the system's TEE codebase and place them in a separate binary file. ❷ **remove "enclave IDs":** since ECalls' normal world counterparts take "enclave ID" as the first parameter, `Function insourcer` re-constructs these callers' parameter lists without the "enclave ID" parameter. ❸ **construct call-chain:** SGX's programming restrictions require that the code in the outside world provide a dedicated pointer to store the values returned by ECalls. Hence, to pass the returned value and construct the call-chain from the outside caller to the extracted ECalls, `Function insourcer` creates a wrapper function to bridge the callers and the extracted ECalls. That is, each caller invokes its wrapper function, which in turn invokes the extracted ECall and returns the result.



```
char *error_des = malloc(100 * sizeof(char));
sgx_status_t status = log_erros(global_eid, error_des);
```

```
int get_airspeed() { ... }

char* log_erros() { ... }
```

```
wrapper_log_erros(error_des);
```

```
void wrapper_log_erros(char* error_des) {
    char * ret = log_erros();
    memcpy(error_des, ret, 100 * sizeof(char));
}
```

```
char* log_erros() { ... }
```

Figure 6: The TEE Insourcing Refactoring

Figure 6 shows an example of the aforementioned insourcing process. The code snippet at the top is the call-chain before insourcing. That is, the normal world counterpart `log_erros` invokes its Ecall (`int log_erros()`), passing `global_eid` ("enclave ID") and `error_des` (the pointer to the returned values) as parameters. The code snippet at the bottom is the re-constructed call-chain after insourcing: in the aforementioned step ❶ (i.e., extract ECalls), the ECall `error_des`, extracted and placed in a separate file to be moved to the outside world; step ❷ removes the caller's `global_eid` parameter (i.e., remove "enclave IDs"); step ❸ creates function `wrapper_log_erros` (i.e., construct call-chain). The first invoked function is the created `wrapper_log_erros` function, which in turn invokes the extracted `log_erros` function, whose returned value is assigned to the caller-provided pointer `error_des`. As `log_erros` is moved from an SGX enclave to the outside world, all the aforementioned invocations take place in the outside world as well. After moving the annotated ECalls to the outside world, `Function Insourcer` removes the "enclave ID" and initialization/cleanup functions if no SGX enclave needs these unneeded parameters and functions.

8. Evaluation

To evaluate TEE-DRUP's correctness, effectiveness, and programming effort, our evaluation seeks to answer the following questions: **Q1. Correctness**: Does our approach correctly detect which variables are sensitive? **Q2. Effectiveness**: How does our approach affect the system's performance? **Q3. Efforts**: How much programming effort it takes to perform each of TEE-DRUP's tasks: collecting/pre-processing information, computing sensitivity levels, designating variables, annotating/insourcing non-sensitive functions?

Table 3: Projects Information

Project	Author Info	Domain	Code-base Info		Variable Number		
			C/C++ file num	LOC	total	after pre-processing	remove tests
GPS Tracker [52]	1 contributor	GPS	76	62746	1993	1298	NA
PAM module [53]	2 contributors	Authentication	4	709	66	28	NA
su-exec [54]	3 contributors	Privileges	1	109	16	13	NA
mkinitcpio-ykfde [55]	5 contributors	Encryption	3	1107	88	79	NA
Spritz Library [56]	4 contributors	Encryption	1	614	138	126	NA
libomron [57]	Nonpolynomial Labs	Health Care	7	1544	166	150	128
ssniper [58]	Technology Services Group (UIUC)	Personal Info (SSN)	12	2421	618	285	253
emv-tools [59]	1 contributor	Bank & Credit	37	9684	1104	995	862

8.1. Environmental Setup

TEE-DRUP's `Info Collector` uses Clang 6.0's *libtooling tool*. `Pre-processor` and `Sensitivity Analyzer` are implemented in Python-2.7. `Taint Analyzer`, `Function Insourcer`, and IAs are integrated with the public release of LLVM 4.0. The TEE is Intel SGX for Linux 2.0 Release. To calculate the word semantic similarity, we use Google's official pre-trained model (Google News corpus with 3 billion running words [60]). All experiments are performed on a Dell workstation, running Ubuntu 16.04, 3.60GHz 8-core Intel i7-7700 CPU, with 31.2 GB memory.

Real-World Scenario and Micro-benchmarks. *a) To evaluate correctness*, we selected real-world, open-source projects that fit the following criteria: 1) must include C/C++ code, as required by SGX, a popular TEE implementation in GitHub; 2) must operate on sensitive data; 3) should have a codebase whose size would not make it intractable to manually check the correctness of TEE-DRUP's designation results. Based on these requirements, we selected 8 open-source projects whose codebases include at most 2000 variables. These 8 evaluation subjects cover diverse security and privacy domains ("Domain" column in Table 3) for evaluating the *correctness* of designating.

b) To evaluate effectiveness and programming effort, we applied TEE-DRUP to the micro-benchmarks comprising implementations of conventional cryptography algorithms (CRC32, DES, RC4, PC1, and MD5), in use in numerous IoT and mobile systems. Typically, cryptographic operations should be placed in the secure world under TEE protection. However, many TEE implementations have already been integrated with their official cryptographic library (e.g., Intel® SGX SSL [61], OP-TEE Cryptographic API [62]), which includes all the standard cryptography algorithms. In fact, secure coding guidelines recommend that programmers use the official cryptographic libraries in their code rather than some custom cryptographic implementations. As a result, using custom conventional cryptography algorithms inside TEE would be considered unnecessary and unsafe. In addition, if a system prioritizes performance over security, placing these custom algorithms into the normal world would be conducive to satisfying the requirements. So, their TEE-based code should be removed along with the rest of their implementation. In our micro-benchmarks, we emulate the first step of getting rid of this unnecessary code, and then use its execution in the normal world for our performance measurements.

8.2. Evaluation Design

1) Correctness: As shown in Table 3, from each project, we extracted all of its variables[13], creating the initial dataset (the "total" column). Then, we pre-processed the initial dataset to remove invalid items and merge duplicated variables (the "after pre-processing" column). After that, we applied TEE-DRUP's sensitivity analysis (5) to determine the sensitivity level of each program variable, with the levels used to designate variables as sensitive or non-sensitive. Finally, we manually checked whether the results are as expected. That is, we requested a volunteer (6+ years C/C++ experience) to manually label all variables' sensitivity for each project (i.e., 1 — sensitive, 0 — unsure, -1 — non-sensitive) and compared the result with what the TEE-DRUP designated as sensitive and non-sensitive variables (see 8.3).

To demonstrate the relationship between *p-tile* and how TEE-DRUP designates variables as sensitive and non-sensitive designation, our evaluation used the 10%, 20%, 30%, 40% and 50% percentages as *p-tile* (discussed in 5.4). That is, TEE-DRUP designates the variables with sensitivity scores in the top *p-tile* as sensitive and the variables with sensitivity scores in the bottom *p-tile* as non-sensitive. We evaluate the correctness of designation for each *p-tile* scenario. We also made use of the project-provided test code in "libomron", "ssniper", and "emv-tools", whose variables are expected to be non-sensitive, to evaluate whether the presence of this test code impacts our correctness results. In other words, running TEE-DRUP on these programs with or without their test code should show dissimilar results (after removing test code, # of variables is in the "remove tests" column).

Metrics and Calculation. Since variables can be labeled as "unsure" during a manual analysis, our evaluation metrics comprise the unsure and miss rates in addition to accuracy, precision, and recall. To calculate accuracy, precision, and recall, we measure the number of true/false positives and negatives [14] among the TEE-DRUP-designated variables (non-designated variables are not used for calculating accuracy, precision, and recall). To calculate the unsure rate, we measure the number of variables volunteer-labeled as "unsure" among the TEE-DRUP-designated variables and all variables. To calculate the miss

[13] To manage the manual labeling effort, we considered only the variables declared in the evaluation subjects' source code, omitting all system and library variables.

[14] true positives/negatives: human-labeled sensitive/non-sensitive variables are designated as sensitive/non-sensitive; false positives: human-labeled non-sensitive variables are designated as sensitive; false negatives: human-labeled sensitive variables are designated as non-sensitive. Note that, human-labeled unsure variables are not counted, which is quantified by "unsure rate."

rate, we measure the number of volunteer-labeled sensitive variables missing among the TEE-DRUP-designated variables.

2) Effectiveness: We applied TEE-DRUP to move the micro-benchmarks back to the normal world. Before the move, the micro-benchmarks' core functions are placed in SGX. We first annotated these functions as non-sensitive, and applied TEE-DRUP's Function Insourcer to migrate them to the normal world. We measured the system's execution overhead before and after the move. Because every time an SGX function is invoked, its caller must initialize the SGX enclave at first (i.e., "initialize_enclave") and destroy the enclave at the end (i.e., "sgx_destroy_enclave"), without loss of generality, we consider the SGX enclave's initialization and destruction operations as unavoidable operations and include them in the system's TEE-based execution performance as a whole.

3) Programming Effort: We estimated the TEE-DRUP-saved programming effort by counting the uncommented lines of code (ULOC) and computing the difference between the amount of code automatically transformed and the number of manually written IAs that it took. That is, TEE-DRUP save programmer effort, as otherwise all code would have to be transformed by hand. Without loss of generality, we assumed that all IAs were default-configured.

8.3. Results

1) Correctness: TEE-DRUP's computing and designating results are discussed in turn next.

Tables 4,5 shows how correctly TEE-DRUP designated sensitive and non-sensitive variables. Overall, among the 55 independent evaluations in 11 different scenarios, TEE-DRUP performed satisfactorily in *accuracy* (lowest:61.7% highest:100%, 37 times > 80%, never < 60%), *precision* (lowest:48.6% highest:100%, 41 times > 80%, 6 times < 60%), and *recall* (lowest:65.9%, highest:100%, 42 times > 80%, never < 60%). The *p-tile* value impacts these metrics: for small p-tiles, e.g., 10%, only the variables with sensitivity scores in top/bottom 10% are designated as sensitive/non-sensitive, resulting in high accuracy, precision, and recall. In contrast, for large p-tiles (e.g., 50%), some variables with relatively lower sensitivity scores are designated as sensitive, while some variables with higher sensitivity scores as non-sensitive. Hence, a large p-tile may lead to high false positives/negatives, lowering accuracy, precision, and recall.

For the miss rate (the "Miss Rate" column): TEE-DRUP's miss rate is negatively correlated to *p-tile*. That is, the larger the *p-tile*, the more variables are designated as sensitive, and fewer sensitive variables missed, yielding lower miss rates. *For the unsure rate (the "Unsure Rate" column):* We evaluate two types of the unsure rate, the unsure rate of all variables (the "all vars" column), and that of TEE-DRUP-designated variables (the "designated vars" column). The former shows the number of variables the volunteer labeled as "unsure" among all variables, which represents the volunteer's understanding level of the evaluation subject. The latter shows the number of variables the volunteer labeled as "unsure" among

Table 4: Correctness - A

Project	**P-tile**	**Accuracy**	**Precision**	**Recall**	**Unsure Rate**		**Miss Rate**
					designated vars	all vars	
GPS Tracker	10%	90.6%	93.8%	96.2%	67.3%	77.2%	70.8%
	20%	89.5%	93.7%	95.0%	70.6%		48.2%
	30%	88.1%	93.0%	94.0%	74.0%		33.1%
	40%	85.3%	91.1%	92.4%	77.1%		24.5%
	50%	84.1%	90.3%	91.4%	77.2%		8.6%
PAM module	10%	100%	100%	100%	0%	21.4%	78.6%
	20%	100%	100%	100%	16.7%		57.1%
	30%	92.3%	100%	88.9%	18.8%		42.9%
	40%	81.2%	100%	75%	27.3%		35.7%
	50%	72.7%	83.3%	71.4%	21.4%		28.6%
su-exec	10%	100%	100%	100%	0%	30.8%	66.6%
	20%	100%	100%	100%	16.7%		33.3%
	30%	100%	100%	100%	12.5%		0%
	40%	100%	100%	100%	20%		0%
	50%	88.9%	75%	100%	30.8%		0%
mkinitcpio-ykfde	10%	100%	100%	100%	50.0%	64.6%	61.1%
	20%	92.3%	90.9%	100%	59.4%		44.4%
	30%	90.0%	94.1%	94.1%	58.3%		11.1%
	40%	87.5%	88.9%	94.1%	62.5%		11.1%
	50%	78.6%	80.0%	88.8%	64.6%		11.1%
Spritz Library	10%	100%	100%	100%	50.0%	46.8%	74.5%
	20%	96.3%	96.0%	100%	46.0%		52.9%
	30%	97.3%	96.8%	100%	51.3%		41.2%
	40%	86.5%	92.1%	89.7%	48.0%		31.4%
	50%	82.0%	93.3%	82.4%	46.8%		17.6%

the TEE-DRUP-designated variables, which shows that the volunteer can refer to the TEE-DRUP-designated sensitive variables when deciding whether an "unsure" variable is sensitive. Overall, the unsure rates of small and straightforward projects (e.g., "pam module" and "su-exec") are relatively lower than those of the complex and large ones. Not surprisingly, the volunteer could easily recognize and correctly label the sensitive variables in small and straightforward projects, but had a harder time performing the same task in larger and more complex systems.

For the test code impact, excluding the test code increases the correctness metric (see rows "with tests" and "no tests" in Table 5). That is, the volunteer labeled all the test code's variables as non-sensitive, but certain test variables' identifiers may mislead TEE-DRUP into designating them as sensitive (e.g.,

Table 5: Correctness - B (with/without tests)

Project	**P-tile**	**Accuracy**	**Precision**	**Recall**	**Unsure Rate**		**Miss Rate**
					designated vars	all vars	
libomron (with tests)	10%	64.7%	57.1%	100%	43.3%	52.0%	69.2%
	20%	66.7%	55.6%	100%	40.0%		42.3%
	30%	61.7%	48.6%	100%	47.8%		34.6%
	40%	64.9%	52.5%	95.5%	52.5%		19.2%
	50%	66.7%	52.1%	96.2%	52.0%		3.8%
libomron (no tests)	10%	93.3%	91.7%	100%	42.3%	60.9%	57.7%
	20%	92.6%	89.5%	100%	48.1%		34.6%
	30%	93.8%	91.3%	100%	57.9%		19.2%
	40%	92.9%	89.3%	100%	58.8%		3.8%
	50%	82.0%	75.8%	96.2%	60.9%		3.8%
ssniper (with tests)	10%	84.8%	79.2%	100%	43.1%	51.6%	77.6%
	20%	88.5%	86.5%	97.0%	54.4%		62.4%
	30%	75.9%	85.4%	77.4%	54.1%		51.8%
	40%	71.2%	78.1%	73.5%	51.3%		41.2%
	50%	62.3%	70.9%	65.9%	51.6%		34.1%
ssniper (no tests)	10%	100%	100%	100%	42.0%	58.1%	76.5%
	20%	97.7%	100%	96.9%	56.9%		63.5%
	30%	85.1%	100%	80.0%	55.9%		52.9%
	40%	80.2%	98.0%	75.8%	57.4%		41.2%
	50%	72.6%	96.7%	68.2%	58.1%		31.8%
emv-tools (with tests)	10%	85.5%	92.0%	88.5%	65.5%	61.8%	69.3%
	20%	75.3%	80.0%	80.0%	61.3%		49.2%
	30%	72.0%	72.4%	76.7%	63.5%		40.7%
	40%	69.7%	63.5%	78.8%	62.3%		28.0%
	50%	65.5%	54.4%	78.7%	61.8%		21.3%
emv-tools (no tests)	10%	91.5%	100%	89.6%	65.7%	71.3%	71.3%
	20%	84.3%	98.7%	81.5%	66.6%		50.0%
	30%	79.2%	95.6%	76.3%	71.2%		42.0%
	40%	74.7%	84.1%	77.9%	71.3%		29.3%
	50%	70.4%	74.2%	78.6%	71.3%		21.3%

variable “key” is tested by the test code in “ssniper”).

2) Effectiveness: Table 6 shows the execution performance of the micro-benchmarks (in microseconds μs). Taking as the baseline the system’s TEE-based performance (“in-TEE” column), automatically moving code to the normal world sharply decreases the execution time (“Move-outside” column). The decreases are due to the eliminated overheads of setting-up/cleaning enclaves and communication between the normal world and TEE. The shorter is a subject's execution time in the normal world, the more pronounced is its performance improvement (e.g., moving back DES to the normal world increased its execution performance by a factor of 10K).

Table 6: Effectiveness (microseconds — μs)

Algorithm	in-TEE	Move-outside
DES	45601.1	2.4
CRC32	41374.9	252.1
MD5	92011.6	68193.4
PC1	50693.1	20190.1
RC4	111412.0	51312.5

3) Programming Effort: Table 7 shows how much programming effort it takes to use TEE-DRUP to move the mirco-benchmarks to the normal world. Since we assumed that all IAs were default-configured, the rest of the subjects in our micro-benchmark suite share similar results. That is, to automatically transform these mirco-benchmarks, developers only add two lines of IAs to annotate per a non-sensitive function. During the transformation, TEE-DRUP generates/transforms about 15 ULOC (including the IR and source code). Finally, a developer needs to clean up the source code to remove the SGX headers (e.g., “Enclave_t.h”). Thus, with TEE-DRUP, developers only need to specify the non-sensitive functions, and manually remove some no-longer used headers. TEE-DRUP performs all the remaining transformation and generation tasks automatically.

Table 7: Programming Effort (ULOC)

Algorithm	IAs	Generate & Transform	Adjust
DES/CRC32/MD5/PC1/RC4	≈2	≈15	≈1

8.4. Discussion

1) Correctness: Based on our results (Tables 4,5), TEE-DRUP shows satisfying *accuracy*, *precision*, and *recall*, but suffers from an unstable *miss rate* (lowest 0%, highest 78.6%). This unstable rate is due to: (a) low p-tile numbers cause TEE-DRUP to designate fewer variables, (b) variable may not be named according to common naming convention (e.g., in “PAM module”, “pw” rather than “pwd” or “password”, designates stored passwords), and (c) some identifiers may not be included from our dictionary of security terms (e.g., because the dictionary does not include ‘ssn”, TEE-DRUP omits variables “ssn*” in “ssniper” as sensitive).

To reduce the *miss rates*, we recommend that developers select a suitable p-tile. In general, the larger the p-tile, the lower the miss rates. However, if the p-tile is too large, too many variables end up designated as sensitive/non-sensitive with their accuracy/precision/recall calculated (the metric calculation is detailed in 8.2-1), causing a low miss rate but a high number of false positives/negatives and low accuracy/precision/recall (row "50%" in Tables 4,5). Also, developers can add additional domain terms (e.g., ssn) to the dictionary, so the corresponding identifiers' sensitivity scores would increase. Besides, to further improve TEE-DRUP's performance, we recommend that developers exclude all testing functionality before analyzing any project.

Overall, TEE-DRUP's satisfactory *accuracy*, *precision*, and *recall* make it a better fit as a recommendation rather than a decision-making system. To help developers analyze realistic systems, TEE-DRUP automatically infers the sensitivity levels for all program variables, designating them as sensitive/non-sensitive. Then, by consulting the resulting sensitivity levels and designations, a developer can verify and confirm which variables are indeed sensitive.

Further, our experiences with TEE-DRUP show that NLP can be effective in determining variable sensitivity. Even with a general NLP model and a common security term dictionary, TEE-DRUP's NLP technique computes variables' sensitivity accurately enough to make it a practical recommendation system. By refining NLP models and term dictionaries, one can increase both the accuracy and applicability of our technique.

2) Effectiveness: Our results illustrate how moving non-sensitive functions to the normal world drastically increases system performance. Hence, TEE-DRUP can improve real-time compliance (e.g., a TEE-based function failing to meet execution deadlines) and help mitigate DoS attacks. Besides, by reducing the attack surface, these moves would also can help mitigate other TEE-based vulnerabilities (e.g., buffer overflows in TEE-based functions). Hence, TEE-DRUP can increase both system performance and security protection.

3) Programming Effort: To move our benchmarks to the normal world, TEE-DRUP requires ≈3 ULOC (i.e., ≈2 for IAs, ≈1 for adjusting). To manually reproduce the code transformation/generation of TEE-DRUP would require modifying ≈15 ULOC. Although this number may seem like a reasonable manual task, it is rife with hidden costs that can be significant. To perform this refactoring correctly requires 1) understanding the source code; 2) manually locating Ecalls, "Enclave id", and the set up/clean functions; 3) ensuring that all the manual transformations are correct. By automatically insourcing code, TEE-DRUP eliminates all these hidden costs.

4) Applicability: Currently, TEE-DRUP works with SGX projects written in C/C++. Although SGX is a popular TEE implementation and C/C++ are SGX-supported programming languages, other TEEs, including Trustonic Kinibi, Huawei's TrustedCore, and Qualcomm's QTEE, have also been in use in domains of Internet of Things (IoT) and Cyber-Physical Systems (CPS) [63, 64]. Furthermore, the analysis of open-source repositories reveals that TEEs are also used with other languages (e.g., RUST [65], Java [66]). Thus, one applicability question is whether TEE-DRUP can be applied or extended to other TEEs and programming languages.

Recall that TEE-DRUP comprises (a) a heuristic for identifying sensitive/non-sensitive data and (b) an automated refactoring for removing non-sensitive code out of TEE. These parts are independent of each other, so we discuss their respective applicability in turn.

- For (a), to support other programming languages, only the variable information collection module (Section 5.1) needs to be modified. Our current solution implements a *libtooling tool* program that extracts variable information (e.g., name, type, file path) from a given source code. However, the *libtooling tool* only supports C/C++. Hence, to extract information from projects written in other languages, we need to apply their specific code analysis tools or frameworks. For example, we can leverage Soot [67] to extract variable information from Java bytecode. Once we obtain this information, the rest of the analysis processes in (a) remain intact for all programming languages. Besides, the specifics of TEE implementations have no bearing on (a)'s variable sensitivity analysis.

- For (b), to support other programming languages, TEE-DRUP needs to apply different LLVM front-ends. TEE-DRUP's current code refactoring process includes taint analysis, program partitioning, and code generation, all of which work with the LLVM intermediate representation (IR) level. Hence, TEE-DRUP applies the LLVM front-ends to convert the original source code into LLVM IR code for further analysis and transformation. Although LLVM's official front-end is Clang for C/C++ only [68], open-source contributors have been creating LLVM-based front-ends (e.g., llgo for Go [69], kaleidoscope for Haskell [70], Rubinius for Ruby [71]). Thus, by leveraging these front-ends, TEE-DRUP can be applied to other programming languages. In this case, a major technical hurdle is that some open-source front-ends may be unreliable, and some languages remain unsupported.

  Moreover, to support interpreted languages or those that require Just-In-Time compilation, they must first execute within a TEE. Specifically, to make it possible, the runtime systems of these languages might need to be customized and placed into a TEE. Note that, these runtime systems would need to be customized for individual TEEs, thus introducing a considerable engineering overhead.

  Similarly, to support other TEEs, TEE-DRUP also needs to adjust its taint analysis and code transformation modules for their particular communication interfaces (e.g., SGX uses `sgx_create_enclave` to initialize the communication session while OP-TEE uses `TEEC_InitializeContext`). The taint analysis module depends on these interfaces to track when and where a caller invokes a TEE function. The code transformation module adapts these interfaces to decide how and where it should remove the unnecessary TEE code after moving each function out. However, accommodating these TEE-specific differences would unduly increase the required engineering effort.

To sum up, it would only take an additional engineering effort to extend our approach to other TEEs and programming languages. Since it would not uncover additional conceptual insights, we chose to focus our efforts on a popular TEE platform, Intel SGX. However, developers can leverage our technical insights to implement similar solutions for other TEEs or languages, taking into account the tradeoffs between utility, reliability, and engineering workload for each implementation scenario.

5) Miscellanea: For TEE-DRUP's toolchain performance: the time taken by program and data analyses tasks is rarely a decisive factor that determines their utility and value, the entire toolchain exhibits acceptable runtime performance. The most time-consuming task—sensitivity computing—takes ≈ 10 minutes for the largest evaluation subject (i.e., GPS Tracker). The remaining TEE-DRUP's tasks complete in seconds.

6) Threats to validity: The internal validity is threatened by our procedure that obtains the ground truth for sensitive variables. In fact, since we evaluate with third-party real-world subjects, it would be unrealistic to expect that our volunteers could designate the variables in these subjects with perfect certainty. To mitigate this threat, we apply the reported "unsure rate" to quantify the reliability of TEE-DRUP's results.

The external validity is threatened by evaluating only eight third-party C/C++ projects. In fact, the major reason hindering us from studying more projects is the need to verify our approach's correctness by hand. In particular, we need to manually check if our approach designates sensitivity for each variable correctly or not. Hence, we only evaluate our approach on eight open-source projects, which cover common sensitive variable scenarios (e.g., location, authentication, encryption, financial account) and include 2000+ variables in total. Enlarging the codebase would increase the number of variables and make it intractable to verify the results manually. To mitigate this threat, we plan to open-source our implementation after publishing the article, so fellow researchers and practitioners could apply it to additional evaluation subjects, with their experiences further validating the practical applicability of our approach.

9. Related work

TEE-DRUP is related to detecting code vulnerabilities, taint analysis, and reducing trusted computing base.

1) Code Vulnerability Detection. Much of prior work focuses on detecting various code vulnerabilities, including bugs, security risks, and anomalous operations.

a) Traditional program analysis: Traditional program analysis, both static and dynamic, has been applied to detect exploitable bugs and other vulnerabilities [72, 73, 74, 75, 76, 77]. By relying on predefined rules or known program properties for detection, static analyses cannot handle *Zero-Day* exploits.

By relying on code instrumentation, dynamic approaches cannot cover all possible inputs and runtime states.

b) Detecting vulnerabilities via machine learning: To mitigate the limitations of tradition program analysis, machine learning techniques have been applied to detect vulnerabilities. By analyzing 183 known vulnerabilities of the Linux kernel and Apache server projects, Younis et al. create a prediction model that identifies which characteristics of vulnerable code make it most exploitable [78]. By combining function embedding and taint analysis, Yamaguchi et al. automatically detect missing security checks in vulnerable codebases [79]. By training a large amount of collected test cases, Grieco et al. build a predictor that detects potentially vulnerable test cases [80]. By mining the encoded information from the discovered vulnerabilities in a large volume of categorized repositories, Sadeghi et al. improve the efficiency of static analysis for detecting vulnerabilities [81].

c) Semantics resolution for detecting vulnerabilities: Since textual information can expose sensitive data, potential security and privacy risks can be detected by resolving sensitive data semantics. Independently implemented SUPOR and UIPicker automatically identify sensitive user input by applying NLP techniques on the extracted UI resources to identify suspicious keywords [26, 27]. UiRef solves the same problem while also resolving ambiguous words [82]. ICONINTENT identifies sensitive UI widgets in Android apps by both resolving textual labels and classifying icons [83].

In summary, to detect vulnerable code or anomalous operations, traditional code analysis techniques rely on pre-defined rules or custom instrumentation. Machine learning techniques either infer rules from regular codebases or identify patterns from known vulnerable code samples. Semantic resolution techniques supplement source code information with that of UI labels. In contrast, TEE-DRUP focuses on data as the origin of vulnerabilities by analyzing variables. It provides an NLP-based sensitivity analysis that identifies sensitive variables in danger of exploitation or leakage. Both the aforementioned approaches and TEE-DRUP suffer from false positives and negatives. However, as a recommendation rather than a decision-making system, TEE-DRUP is not as impacted by these problems. It identifies and presents potentially sensitive variables to developers, who ultimately decide which variables must be protected in TEE.

2) Taint Analysis. TaintDroid [29] tracks the data flow in Android apps, with many related techniques developed to detect code vulnerabilities in IoT and mobile systems [28, 26, 31, 84, 30, 85, 86]. JN-SAF enhances taint analysis to track inter-language dataflow in Android apps, whose codebase contains both Java and native (C/C++) files [87]. However, no existing taint-based techniques focus on capturing tainted dataflow in systems that rely on TEE. By mapping the TEE-based functions to their callers in the outside world, TEE-DRUP extends taint analysis to TEE-based systems.

3) Reduce Trusted Computing Base (TCB) Singaravelu et al. identify the problem of large TCBs and how to reduce them in three real-world applications [88]. Lind et al. present an automated partition tool, Glamdring, that partitions a system by placing only the sensitive data and code in the TEE [16]. Similarly, RT-Trust and Ptrsplit automatically move developer-specified functions to TEE to reduce TCB [11, 14]. Qian et al. reduce the size of deployed binaries by developing RAZOR, an automatic code-debloating tool [89]. Rubinov et al. use FlowDroid's taint analysis to track developer-specified data, so only the relevant functions can be moved to TEE [13].

In general, TEE-DRUP differs from these prior approaches, which focus on moving "sensitive" code and data into TEE (i.e., forward direction). In contrast, it focuses on 1) identifying (non-)sensitive data and code and 2) moving non-sensitive code out of TEE (i.e., backward direction). Both of these refactoring techniques are designed to place the necessary code and data in the correct environments (i.e., inside or outside TEE). In particular, based on our findings, oftentimes code is placed in TEE unnecessarily, thereby necessitating the TEE-DRUP's backward direction refactoring. By deploying TEE-DRUP, developers can discover and remove the unnecessary TEE code, thereby reducing the TCB's size and improving system performance. Drilling down to specific technical details, some prior approaches (e.g., Glamdring) work at the source code level. Given multiple source code files, these approaches have to analyze each code file one by one, or merge several files into a single one. Also, they need to handle the dependencies between different source code files. In contrast, TEE-DRUP works at the LLVM intermediate representative (IR) level. Since IR code can be easily combined via LLVM's toolchain, TEE-DRUP can analyze a combined single IR code file, including all the given project information.

4) Enable Unmodified Program to Run inside TEE. SCONE [90], SGX-LKL [91], and Gramine (formerly Graphene) [92] provide a small but featured library OSes running inside TEE, enabling unmodified applications to be directly executed in TEE under protection. These approaches eliminate the need for code transformation/generation to move it in and out of TEE. However, the problem of identifying which code is sensitive and needs to be protected in TEE does not go away. Hence, our variable sensitivity analysis remains applicable. Besides, when an entire project is placed in TEE, the resulting performance overhead can be high. For example, under SGX, code inside enclaves cannot make system calls, thus slowing down dependent functionalities (although SCONE mitigates this issue, it still cannot match the performance of running the code outside the enclave). A promising future work direction would be to study the performance characteristics of these TEE-based library OSes.

10. Conclusions

In this article, we investigated the status quo of using TEE in real-world software development by conducting the first study of over 400 open-source TEE-related projects. Our study uncovered that (1) developers frequently overuse TEE by isolating code and data that are not sensitive, and (2) a considerable number of projects put more than a half of their entire codebase into TEE. Motivated by these findings, we created TEE-DRUP, an semi-automated toolchain whose program analysis and transformation techniques (1) designate sensitive variables in a given codebase via NLP-based variable sensitivity analysis; (2) discover non-sensitive functions via a TEE-aware taint analysis; (3) automatically refactor binary code by moving the non-sensitive functions out of TEE. By applying TEE-DRUP, developers can reduce TCB, thus both improving system performance and decreasing attack surface. Our evaluation shows that TEE-DRUP effectively and correctly performs these program analyses and transformations, thus improving system security/privacy, increasing performance (with the speedup ranging between 1.35 and 10K), and saving programmer effort.

References

[1] Y. Liu, E. Tilevich, Reducing the price of protection: Identifying and migrating non-sensitive code in tee, in: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), IEEE, 2020, pp. 112–120.

[2] Vormetric Data Security, Vormetric data security, trends in encryption and data security. cloud, big data and iot edition, vormetric data threat report., https://dtr.thalesesecurity.com/pdf/cloud-big-data-iot-2016-vormetric-dtr-deck-v2.pdf (2016).

[3] CVE-2020-7964., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7964 (2020).

[4] CVE-2020-5202., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5202 (2020).

[5] CVE-2019-7888., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7888 (2019).

[6] CVE-2019-6655., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6655 (2019).

[7] CVE-2019-3901., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3901 (2019).

[8] CVE-2019-17590., https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17590 (2019).

[9] V. Costan, S. Devadas, Intel SGX explained., IACR Cryptology ePrint Archive 2016 (086) (2016) 1–118.

[10] OP-TEE, Open portable trusted execution environment, https://www.op-tee.org/ (2019).

[11] Y. Liu, K. An, E. Tilevich, RT-Trust: automated refactoring for trusted execution under real-time constraints, in: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, 2018, pp. 175–187.

[12] N. Weichbrodt, P.-L. Aublin, R. Kapitza, SGX-perf: A performance analysis tool for intel sgx enclaves, in: Proceedings of the 19th International Middleware Conference, 2018, pp. 201–213.

[13] K. Rubinov, L. Rosculete, T. Mitra, A. Roychoudhury, Automated partitioning of Android applications for trusted execution environments, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 923–934.

[14] S. Liu, G. Tan, T. Jaeger, Ptrsplit: Supporting general pointers in automatic program partitioning, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2359–2371.

[15] A. Senier, M. Beck, T. Strufe, Prettycat: Adaptive guarantee-controlled software partitioning of security protocols, arXiv preprint arXiv:1706.04759.

[16] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al., Glamdring: Automatic application partitioning for intel SGX, in: 2017 USENIX Annual Technical Conference, 2017, pp. 285–298.

[17] M. Ye, J. Sherman, W. Srisa-an, S. Wei, Tzslicer: Security-aware dynamic program slicing for hardware isolation, in: 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), IEEE, 2018, pp. 17–24.

[18] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, D. E. Porter, Cooperation and security isolation of library oses for multi-process applications, in: Proceedings of the Ninth European Conference on Computer Systems, ACM, 2014, p. 9.

[19] A. Baumann, M. Peinado, G. Hunt, Shielding applications from an untrusted cloud with haven, ACM Transactions on Computer Systems (TOCS) 33 (3) (2015) 8.

[20] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al., SCONE: Secure linux containers with Intel SGX, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 689–703.

[21] S. C. Misra, V. C. Bhavsar, Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: International Conference on Computational Science and Its Applications, Springer, 2003, pp. 724–732.

[22] A. Biondo, M. Conti, L. Davi, T. Frassetto, A.-R. Sadeghi, The guard's dilemma: Efficient code-reuse attacks against intel {SGX}, in: 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 1213–1227.

[23] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, B. B. Kang, Hacking in darkness: Return-oriented programming against secure enclaves, in: 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 523–539.

[24] SANS, Glossary of security terms, `https://www.sans.org/security-resources/glossary-of-terms/` (2019).

[25] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, W. G. Halfond, Recdroid: automatically reproducing android application crashes from bug reports, in: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, 2019, pp. 128–139.

[26] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, G. Jiang, {SUPOR}: Precise and scalable sensitive user input detection for android apps, in: 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 977–992.

[27] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, X. Wang, Uipicker: User-input privacy identification in mobile applications, in: 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 993–1008.

[28] Y. Hu, O. Riva, S. Nath, I. Neamtiu, Elix: Path-selective taint analysis for extracting mobile app links, in: Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, ACM, 2019, pp. 193–206.

[29] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, ACM Transactions on Computer Systems (TOCS) 32 (2) (2014) 5.

[30] M. Sun, T. Wei, J. Lui, Taintart: A practical multi-level information-flow tracking system for android runtime, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 331–342.

[31] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: Detecting inter-component privacy leaks in android apps, in: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, 2015, pp. 280–291.

[32] llvm-admin team, The LLVM Compiler Infrastructure, `https://llvm.org/` (2019).

[33] The Clang Team, LibTooling, `https://clang.llvm.org/docs/LibTooling.html` (2019).

[34] llvm-admin team, LLVM Pass, `http://llvm.org/docs/WritingAnLLVMPass.html` (2019).

[35] Google, Google C++ style guide, `https://google.github.io/styleguide/cppguide.html` (2019).

[36] IBM, Naming standards, `https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.7.0/com.ibm.swg.im.iis.ia.application.doc/topics/c_naming_stds.html` (2019).

[37] Microsoft, Naming guidelines, `https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines` (2019).

[38] Google, What to look for in a code review, `https://google.github.io/eng-practices/review/reviewer/looking-for.html` (2019).

[39] Hashnode online discussion, Is it normal for a programmer to forget a programming language over time?, `https://hashnode.com/post/ is-it-normal-for-a-programmer-to-forget-a-programming-language-over-time cj9i2ojbu02dxw5wt73ain1rs` (2017).

[40] GitHub search api, https://developer.github.com/v3/search/search-repositories (2020).

[41] Mhoekstr, Intel® SGX for dummies (Intel® SGX design objectives), Intel® Software.
URL `https://tinyurl.com/yh7n6rnl`

[42] M. Sabt, M. Achemlal, A. Bouabdallah, Trusted execution environment: what it is, and what it is not, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 1, IEEE, 2015, pp. 57–64.

[43] Adventures of an enclave (SGX / TEEs), By Leland Lee.
URL `https://hackernoon.com/adventures-of-an-enclave-sgx-tees-9e7f8a975b0b`

[44] Roskakori, Pygount, https://github.com/roskakori/pygount.

[45] Intel, SGX sample code, `https://github.com/intel/linux-sgx/tree/master/SampleCode` (2019).

[46] Y. Liu, E. Tilevich, VarSem: Declarative expression and automated inference of variable usage semantics, in: Proceedings of 19th International Conference on Generative Programming: Concepts Experiences.

[47] Google, word2vec, `https://code.google.com/archive/p/word2vec/` (2019).

[48] P. K. Sahoo, S. Soltani, A. K. Wong, A survey of thresholding techniques, Computer vision, graphics, and image processing 41 (2) (1988) 233–260.

[49] Intel, Intel software guard extensions (intel sgx) sdk for linux, https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf (2018).

[50] C. Team, Attributes in clang, https://clang.llvm.org/docs/AttributeReference.html (2018).

[51] GNU, Using the gnu compiler collection (gcc), http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html (2018).

[52] GPS/GNSS Tracker for STM32Primer2, https://github.com/nemuisan/STM32Primer2_GPS_Tracker (2019).

[53] PAM module, https://github.com/tiwe-de/libpam-pwdfile (2019).

[54] su-exec, https://github.com/ncopa/su-exec (2019).

[55] Full disk encryption with Yubikey (Yubico key), https://github.com/eworm-de/mkinitcpio-ykfde (2019).

[56] Spritz Library For Arduino, https://github.com/abderraouf-adjal/ArduinoSpritzCipher (2019).

[57] libomron, https://github.com/openyou/libomron (2019).

[58] Ssniper Social Security Scanner for Linux, https://github.com/racooper/ssniper (2019).

[59] Tools to work with EMV bank cards, https://github.com/lumag/emv-tools (2019).

[60] Google, word2vec-GoogleNews-vectors, https://github.com/mmihaltz/word2vec-GoogleNews-vectors (2019).

[61] Intel Software Guard Extensions SSL, https://github.com/intel/intel-sgx-ssl (2021).

[62] OP-TEE Cryptographic implementation, https://github.com/OP-TEE/optee_docs/blob/master/architecture/crypto.rst (2021).

[63] M. Busch, J. Westphal, T. Mueller, Unearthing the trustedcore: A critical review on huawei's trusted execution environment, in: 14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20), 2020.

[64] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, E. Conchon, Secure and trusted execution: Past, present, and future-a critical review in the context

of the internet of things and cyber-physical systems, in: 2016 IEEE Trustcom/BigDataSE/ISPA, IEEE, 2016, pp. 168–177.

[65] Fortanix Rust Enclave Development Platform, https://github.com/fortanix/rust-sgx (2021).

[66] The public parts of R3's software, https://github.com/domsteil/sgxjvm-public (2021).

[67] Soot, https://github.com/soot-oss/soot (2021).

[68] Clang, https://github.com/llvm-mirror/clang (2021).

[69] llgo, https://github.com/go-llvm/llgo (2021).

[70] Haskell LLVM, https://github.com/sdiehl/kaleidoscope (2021).

[71] The Rubinius Language Platform, https://github.com/rubinius/rubinius (2021).

[72] D. Evans, D. Larochelle, Improving security using extensible lightweight static analysis, IEEE software 19 (1) (2002) 42–51.

[73] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, R. Venkatapathy, Righting software, IEEE software 21 (3) (2004) 92–100.

[74] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE software 25 (5) (2008) 22–29.

[75] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later: using static analysis to find bugs in the real world, Communications of the ACM 53 (2) (2010) 66–75.

[76] P. Godefroid, Random testing for security: blackbox vs. whitebox fuzzing, in: Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), ACM, 2007, pp. 1–1.

[77] P. Godefroid, M. Y. Levin, D. Molnar, Sage: whitebox fuzzing for security testing, Communications of the ACM 55 (3) (2012) 40–44.

[78] A. Younis, Y. Malaiya, C. Anderson, I. Ray, To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, ACM, 2016, pp. 97–104.

[79] F. Yamaguchi, C. Wressnegger, H. Gascon, K. Rieck, Chucky: Exposing missing checks in source code for vulnerability discovery, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 499–510.

[80] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, L. Mounier, Toward large-scale vulnerability discovery using machine learning, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, ACM, 2016, pp. 85–96.

[81] A. Sadeghi, N. Esfahani, S. Malek, Mining the categorized software repositories to improve the analysis of security vulnerabilities, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2014, pp. 155–169.

[82] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, T. Xie, Uiref: analysis of sensitive user inputs in android applications, in: Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, ACM, 2017, pp. 23–34.

[83] X. Xiao, X. Wang, Z. Cao, H. Wang, P. Gao, Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps, in: Proceedings of the 41st International Conference on Software Engineering, IEEE Press, 2019, pp. 257–268.

[84] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: Acm Sigplan Notices, Vol. 49, ACM, 2014, pp. 259–269.

[85] J. Schütte, D. Titze, J. M. De Fuentes, Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps, in: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2014, pp. 370–379.

[86] L. Li, T. F. Bissyandé, D. Octeau, J. Klein, Droidra: Taming reflection to support whole-program analysis of android apps, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 318–329.

[87] F. Wei, X. Lin, X. Ou, T. Chen, X. Zhang, Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2018, pp. 1137–1150.

[88] L. Singaravelu, C. Pu, H. Härtig, C. Helmuth, Reducing tcb complexity for security-sensitive applications: Three case studies, in: ACM SIGOPS Operating Systems Review, Vol. 40, ACM, 2006, pp. 161–174.

[89] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, W. Lee, {RAZOR}: A framework for post-deployment software debloating, in: 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1733–1750.

[90] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, C. Fetzer, SCONE: Secure linux containers with intel SGX, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 689–703. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[91] SGX-LKL, https://github.com/lsds/sgx-lkl (2021).

[92] Gramine, https://github.com/gramineproject/gramine (2021).