Reducing the Price of Protection: Identifying and Migrating Non-Sensitive Code in TEE

Yin Liu
Software Innovations Lab, Virginia Tech
yinliu@cs.vt.edu

Eli Tilevich
Software Innovations Lab, Virginia Tech
tilevich@cs.vt.edu

***Abstract*—As the trusted computing base (TCB) unnecessarily increases its size, the performance and security of Trusted Execution Environments (TEE) can deteriorate rapidly. Existing solutions focus on placing only the necessary program parts in TEE, but neglect the numerous cases of legacy software with misplaced TEE-based non-sensitive code. In this paper, we introduce a new type of software refactoring—*TEE Insourcing*—that identifies and migrates non-sensitive code out of TEE. We present TEE-DRUP, the first semi-automated *TEE Insourcing* framework whose process comprises two phases: (1) a variable sensitivity analysis designates each variable as sensitive or non-sensitive; (2) a compiler-assisted program transformation automatically moves the functions that never operate on the sensitive variables out of TEE. Developers can participate to verify and confirm sensitive variables, and specify additional non-sensitive functions to migrate. The evaluation results of TEE-DRUP on real-world programs are encouraging. TEE-DRUP distinguishes between sensitive and non-sensitive variables with satisfactory accuracy, precision, and recall — all of their actual values are greater than 80% in the majority of evaluation scenarios. Further, moving non-sensitive code out of TEE improves system performance, with the speedup ranging between 1.35 and 10K. Finally, TEE-DRUP's automated program transformation requires only a small programming effort.**

***Index Terms*—TEE, Security Analysis and Transformation**

I. 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 [1]. Sensitive data and code processing them are the target of many data disclosure and code tampering attacks [2]–[7]. An increasingly popular protection mechanism isolates sensitive code and data from the outside world[1] in a trusted execution environment (TEE) (e.g., SGX [8] and OP-TEE [9]). However, as increasing volumes of code run in TEE, not all of that code is sensitive, so the trusted computing base (TCB) grows 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 [10]. For example, numerous function invocations, entering or leaving TEE, trigger a large volume of in/out communication, slowing down the entire system [11]. When it comes to security, prior works [10], [12]–[16] move programmer-specified data or functions, even the entire system (e.g., Graphene [17], Haven [18], and SCONE [19]) 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 [20], 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[2] [21], [22]. One—thus far overlooked—approach that can increase the performance and reduce the attack surface of TEE-based execution is to move the unnecessary code (i.e., non-sensitive code) from the TEE to the outside world.

To address this problem, 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 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. 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.

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 this designations, developers then verify and confirm which variables are indeed sensitive. In phase (2), via a declarative meta-programming model, a compiler-assisted program analysis and transformation (a) identifies those TEE-based functions that never operate on developer-confirmed sensitive variables as *non-sensitive functions*; developers then confirm the ones to move to the outside world; (b) 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

[1]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 § II-A).

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

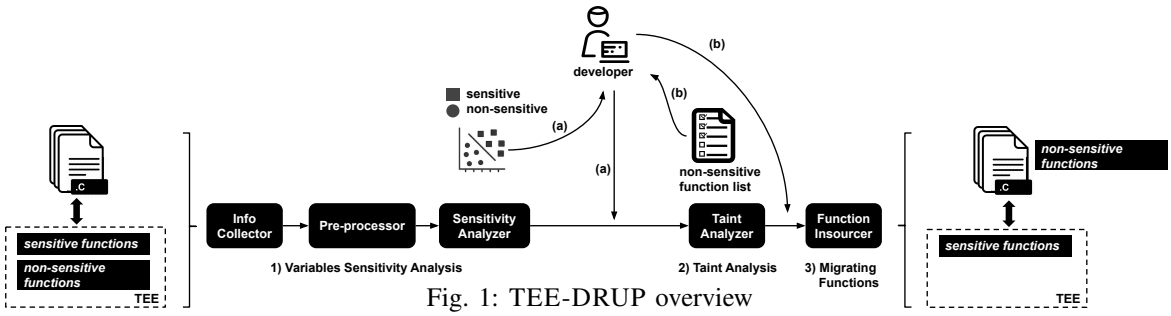

Fig. 1: TEE-DRUP overview

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 paper is as follows:

1) ***TEE Insourcing***, a novel approach to reducing the TCB size, concretely realized as TEE-DRUP that offers:
   - *a variable sensitivity analysis* that designates program variables as sensitive or non-sensitive by using NLP, assisting developers in verifying and confirming (non-)sensitive variables.
   - *a compiler-assisted automated program transformation* that moves TEE-based non-sensitive functions to the outside world to satisfy various requirements.
2) **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.

II. Background & Solution Overview

We first introduce the technical background, and then explain TEE-DRUP's application by example.

A. Definitions and Enabling Technologies

Sensitive & non-Sensitive: "Sensitive" describes all security-related objects (e.g., passwords, 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[3] refers to as "security terms" [23]. *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."
Intel's Software Guard Extensions (SGX) [8]: To protect the integrity and confidentiality of sensitive code and data, 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/implementation focuses on SGX, due to the maturity of SGX implementation.
Natural language processing (NLP): NLP techniques for processing natural languages have been used for identifying security information in program code [24]–[26]. TEE-DRUP's NLP-based sensitivity analysis designates sensitive variables that should be protected in TEE.
LLVM *[27]* is a mature compiler infrastructure used for program analysis at the source-code and binary levels. For source-code analysis, LLVM features `libtooling tool` [28], while for binary analysis, it features `Pass`. 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 moves non-sensitive functions outside TEE.

B. TEE-DRUP *Process*

Figure 1 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 (e.g., name, type, path) and designates sensitive variables. With the designated variables at their disposal, developers then verify and confirm which variables are indeed sensitive (step-a).

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 (step-b).

Phase (3) applies `Function Insourcer` to automatically adjust the TEE-related call interfaces, remove their

[3]An authoritative source for information security certification/research.

TEE metadata[4], 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.

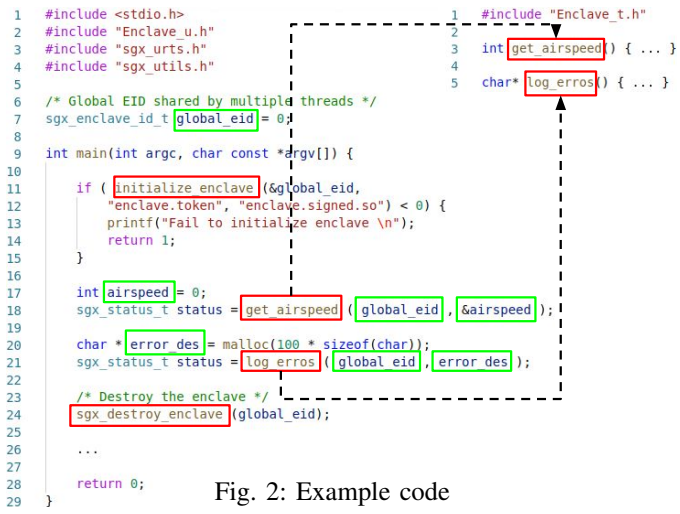

Fig. 2: Example code

C. 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 [29], with the side-by-side code snippets appearing in Figure 2. 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 21). Further, to obtain ECalls' return value, a pointer should be provided as another extra parameter (lines 17,20). Moreover, initialization (`initialize_enclave`, line11) and cleanup (`sgx_destroy_enclave`, line24) functions must be called both before and after interacting with a TEE-based function.

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[5], 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 outside TEE. `Function Insourcer` extracts `log_erros` into a program unit executing outside the enclave, and redirects its callers to invoke the extracted code instead[6].

D. 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, IBM, and Microsoft, have established coding conventions requiring that program identifiers be named intuitively [30]–[32]. Regular code reviews often come up with suggestions how to rename identifiers to more meaningfully reflect their roles and usage scenarios [33]. *In addition, since major TEE implementations (e.g., SGX and OP-TEE) only work with the C/C++ language, we only support C/C++ projects.* We plan to extend this support to managed and multi language projects as a future work direction.

III. Analyzing Variables Sensitivity

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

A. Collecting Information

1) Extracting Program Data. `Info Collector` traverses the given system's abstract syntax tree (AST) to locate variable nodes and collect their textual information (i.e., the variable's name, its type's name, its enclosing function's name, and the source file's path). *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. Table I shows an encoded record for variable `const int * the_password`.

TABLE I: Data Format: `const int * the_password;`

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

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

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

[6]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 outside world.

B. Pre-processing

1) Filtering Records. Since in a realistic system, not all variables need to be analyzed, `Pre-processor` identifies and removes those variable records that are used exclusively within SGX enclaves (e.g., enclave IDs) and whose symbolic names are too short (e.g., variables named `i` or `j`). In addition, `Pre-processor` merges duplicate records. *2) Splitting Identifiers.* Since variables' textual information can follow dissimilar naming conventions, such as delimiter-separated (e.g., the_pass_word) or letter case-separated (e.g., thePassword), 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.* Since some parts of identifiers carry no useful sensitivity information (e.g., "the" in "the_password"), `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.

C. 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 [34]:

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) Composite data type (e.g., struct/class/union) can indicate variables' sensitivity. For example, as our evaluation demonstrates (§ V), 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 info: 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 info 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)
λ (i.e., the attenuation rate for file paths)
Output: variables with sensitivity levels

```
1  Function: get_similarity(word_array, dict, λ):
2  sim ← 0
3  foreach word : word_array do
4      txt ← find_most_similar(word, dict)
5      d ← similarity(word, txt)
6      increase sim by d * λ
7  end
8  avg ← average(sim)
9  return avg
10 Function: calculating_main(textual_info_list, dict):
11 foreach var : textual_info_list do
       /* for variable name.                    */
12     var_name ← get_var_name(var)
13     sim_var ← get_similarity(var_name,dict, 1)
       /* for type name.                        */
14     type_name ← get_type_name(var)
15     sim_type ← get_similarity(type_name,dict, 1)
       /* for function name.                    */
16     func_name ← get_func_name(var)
17     sim_func ← get_similarity(func_name,dict, 1)
       /* for file path.                        */
18     path ← get_path(var)
19     sim_path ← get_similarity(path,dict, 0.8)
20     var.sensitivity ←
         (sim_var + sim_func * 0.8 + sim_type * 0.8 + sim_path * 0.5)
21 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 § II-C, consider how `Sensitivity Analyzer` would calculate the sensitivity levels for variables in the input textual-info list. The textual-info list contains variable `error_des` with type `struct ReportInfo`, accessed in function `report_for`, defined in "report/log.c". `Pre-processor` creates the arrays of `error_des`'s name, type, function, and file path to [`error, des`], [`report, info`], [`report`], and [`report, log`], respectively. After that, `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 "excep-

tion") 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 level.

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: $1 * 80\%$).

4) Implementing Sensitivity Analysis. Our algorithm computes the similarity with *Word2vec*, a Google's word embedding tool [35]. 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 [23]. 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 § V.

D. 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 [36]. 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%, 30%, 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.

IV. Insourcing TEE-based Functions

We first discuss how TEE-DRUP identifies which functions to insource, and then describe the incourcing process:

1) Identifying non-sensitive functions. (a) Given TEE-DRUP-designated sensitive variables, developers then manually confirm and mark which ones are indeed sensitive and warrant TEE protection. To mark these variables, TEE-DRUP provides a custom annotation, `sens`. Given the developer-annotated sensitive variables, TEE-DRUP then applies taint analysis to automatically identify those TEE-based functions that reference none of these variables, and as such should be moved out of TEE to the normal world. (b) Given TEE-DRUP-identified non-sensitive functions, developers confirm which ones of them to insource via another custom annotation, `nonsens`. This annotation signals to `Function Insourcer` which functions to extract and migrate from the secure world to the normal world. These custom annotations are referred to as Insourcing Annotation (IA), designed and implemented to follow the *Clang annotation scheme* [37] and GNU style [38].

2) 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.

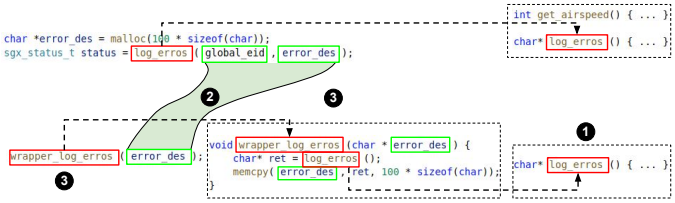

Fig. 3: The TEE Insourcing Refactoring

Figure 3 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 parameters and functions.

V. Evaluation

Our evaluation seeks to answer the following questions: **Q1. Correctness**: Does our approach correctly identify sensitive variables? **Q2. Effectiveness**: How does our approach affect the system's performance? **Q3. Efforts**: How much programming effort it takes to perform TEE-DRUP tasks?

A. Environmental Setup

TEE-DRUP's `Info Collector` uses Clang 6.0's *libtooling tool*. `Preprocessor` 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 [39]). 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 & 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; 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 C/C++ projects whose codebases include at most 2K variables, which covers diverse security and privacy domains ("Domain" column in Table II).

b) To evaluate effectiveness and programming effort, we applied TEE-DRUP to the micro-benchmarks used in several related works, concerned with introducing TEE protection [40], [41]. These micro-benchmarks comprise implementations of commonly used cryptography algorithms (CRC32, DES, RC4, PC1, and MD5), in use in numerous IoT and mobile systems.

TABLE II: Projects Information

Project	Domain	Code-base Info		Variable Number		
		file num	LoC	total	pre-proc	no tests
GPS Tracker [42]	GPS	76	62746	1993	1298	NA
PAM module [43]	Authentication	4	709	66	28	NA
su-exec [44]	Privileges	1	109	16	13	NA
mkinitcpio-ykfde [45]	Encryption	3	1107	88	79	NA
Spritz Library [46]	Encryption	1	614	138	126	NA
libomron [47]	Health Care	7	1544	166	150	128
ssniper [48]	Personal Info (SSN)	12	2421	618	285	253
emv-tools [49]	Bank & Credit	37	9684	1104	995	862

B. Evaluation Design

1) Correctness: As shown in Table II, from each project, we extracted all of its variables[7], creating the initial dataset (the "total" column). Then, we pre-processed the initial dataset to remove invalid items and merge duplicated variables (the "pre-proc" column). After that, we applied TEE-DRUP's sensitivity analysis to determine the sensitivity level of each program variable, with the levels used to designate variables as sensitive or non-sensitive. Finally, 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/non-sensitive variables.

To demonstrate the relationship between *p-tile* and how TEE-DRUP designates variables as sensitive/non-sensitive, our evaluation used the 10%, 30%, and 50% percentages as *p-tile* (see § III-D), and evaluated 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. That is, 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 "no tests" column).

Metrics & 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/recall. To calculate accuracy/precision/recall, we measure the number of true/false positives and negatives[8] among the TEE-DRUP-designated variables (non-designated variables are not used for calculating accuracy/precision/recall). For the unsure rate, we measure the number of variables volunteer-labeled as "unsure" among the TEE-DRUP-designated variables and all variables. For the miss rate, we measure the number of volunteer-labeled sensitive variables missing among the TEE-DRUP-designated variables.

2) Effectiveness: We annotated the micro-benchmarks' major functions placed in TEE as non-sensitive, and applied TEE-DRUP's `Function Insourcer` to move them back to the normal world. We measured the system's execution overhead before and after the move.

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.

C. Results

1) Correctness: Table III shows how correctly TEE-DRUP designated sensitive/non-sensitive variables. Overall, among the 33 independent evaluations in 11 different scenarios, TEE-DRUP performed satisfactorily in ***accuracy*** (lowest:61.7% highest:100%, 21 times>80%, never<60%), ***precision*** (lowest:48.6% highest:100%, 23 times>80%, 3 times<60%), and ***recall*** (lowest:65.9%, highest:100%, 25 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,

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

[8]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."

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): (a) the *unsure rate* of all variables (the "all vars" column) shows the number of variables the volunteer labeled as "unsure" among all variables, which represents the volunteer's understanding level of the evaluation subject. (b) the *unsure rate* of TEE-DRUP-designated variables (the "designated vars" column) shows the number of variables the volunteer labeled as "unsure" among the TEE-DRUP-designated variables, which represents that a developer 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","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 III). 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., variable "key" is tested by the test code in "ssniper").

2) Effectiveness: Table IV 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).

3) Programming Effort: Table V 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 III: Correctness

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%
	30%	88.1%	93.0%	94.0%	74.0%		33.1%
	50%	84.1%	90.3%	91.4%	77.2%		8.6%
PAM module	10%	100%	100%	100%	0%	21.4%	78.6%
	30%	92.3%	100%	88.9%	18.8%		42.9%
	50%	72.7%	83.3%	71.4%	21.4%		28.6%
su-exec	10%	100%	100%	100%	0%	30.8%	66.6%
	30%	100%	100%	100%	12.5%		0%
	50%	88.9%	75%	100%	30.8%		0%
mkinitcpio-ykfde	10%	100%	100%	100%	50.0%	64.6%	61.1%
	30%	90.0%	94.1%	94.1%	58.3%		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%
	30%	97.3%	96.8%	100%	51.3%		41.2%
	50%	82.0%	93.3%	82.4%	46.8%		17.6%
libomron (with tests)	10%	64.7%	57.1%	100%	43.3%	52.0%	69.2%
	30%	61.7%	48.6%	100%	47.8%		34.6%
	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%
	30%	93.8%	91.3%	100%	57.9%		19.2%
	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%
	30%	75.9%	85.4%	77.4%	54.1%		51.8%
	50%	62.3%	70.9%	65.9%	51.6%		34.1%
ssniper (no tests)	10%	100%	100%	100%	42.0%	58.1%	76.5%
	30%	85.1%	100%	80.0%	55.9%		52.9%
	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%
	30%	72.0%	72.4%	76.7%	63.5%		40.7%
	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%
	30%	79.2%	95.6%	76.3%	71.2%		42.0%
	50%	70.4%	74.2%	78.6%	71.3%		21.3%

TABLE IV: 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

TABLE V: Programming Effort (ULOC)

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

D. Discussion

1) Correctness: Based on our results (Table III), 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 § V-B-1), causing a low miss rate but a high number of false positives/negatives and low accuracy/-precision/recall (row "50%" in Table III). 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.

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 function failing to meet execution deadlines). Besides, by reducing the attack surface, these moves would also mitigate other TEE-based vulnerabilities (e.g., buffer overflows in the secure world).

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 significant hidden costs: (a) understanding the source code; (b) manually locating Ecalls, "Enclave id", and the set up/clean functions; (c) ensuring that all the manual transformations are correct. By automatically insourcing code, TEE-DRUP eliminates all these costs.

4) Utility & Applicability: This work presents both a heuristic for identifying sensitive/non-sensitive data and an automated refactoring for moving out non-sensitive code out of TEE. These contributions are independent of each other. It would be impossible to create a heuristic that identifies sensitive/non-sensitive data with perfect precision, as it would require ascertaining the actual program semantics with respect to security and privacy. Hence, TEE-DRUP's designation is intended to simply assist developers in deciding which code are sensitive. Even in the absence of a perfectly precise designation heuristic, the automated *TEE Insourcing* refactoring presents value to the developer by automating the tedious and error-prone program transformations required to refactor out non-sensitive code, even if developers identified such code by hand.

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. 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 results. The *external validity* is threatened by evaluating with only eight third-party C/C++ projects and five cryptography micro-benchmarks used in prior related works. Although used in real projects and containing a wealth of sensitive variable scenarios, our evaluation subjects cannot possibly encompass all possible cases that involve sensitive variables. We plan to mitigate this threat by open-sourcing TEE-DRUP, so fellow researchers and practitioners could apply it to additional evaluation subjects.

VI. Related work

1) 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 [25], [26]. UiRef solves the same problem while also resolving ambiguous words [50]. ICONINTENT identifies sensitive UI widgets in Android apps by both resolving textual labels and classifying icons [51]. Different from these techniques, TEE-DRUP focuses on data as the origin of vulnerabilities by identifying sensitive variables.

2) Reduce Trusted Computing Base (TCB): Singaravelu et al. identify the problem of large TCBs and how to reduce them in three real-world applications [52]. Lind et al. present an automated partition tool, Glamdring, that partitions a system by placing only the sensitive data and code in the TEE [15]. Similarly, RT-Trust and Ptrsplit automatically move developer-specified functions to TEE to reduce TCB [10], [13]. Qian et al. reduce the size of deployed binaries by developing RAZOR, an automatic code-debloating tool [53]. Rubinov et al. use FlowDroid's taint analysis to track developer-specified data, so only the relevant functions can be moved to TEE [12]. TEE-DRUP differs from these prior approaches by focusing on (a) assisting developers in determining which sensitive data and code should be protected in the TEE; (b) automatically moving non-sensitive code to the outside world.

In summary, both the aforementioned approaches and TEE-DRUP suffer from false positives/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.

VII. Conclusion

We presented TEE-DRUP, a semi-automated toolchain that helps developers analyze realistic systems by automatically designating program variables as sensitive/non-sensitive, with satisfactory *accuracy*, *precision*, and *recall*. Developers then confirm which variables are indeed sensitive. TEE-DRUP's automated refactoring reduces TCB, thus both improving system performance and decreasing attack surface.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This research is supported by NSF through the grants #1650540 and #1717065.

REFERENCES

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

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

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

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

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

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

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

[8] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

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

[10] Y. Liu, K. An, and 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*. ACM, 2018, pp. 175–187.

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

[12] K. Rubinov, L. Rosculete, T. Mitra, and 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.

[13] S. Liu, G. Tan, and 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.

[14] A. Senier, M. Beck, and T. Strufe, "Prettycat: Adaptive guarantee-controlled software partitioning of security protocols," *arXiv preprint arXiv:1706.04759*, 2017.

[15] 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.

[16] M. Ye, J. Sherman, W. Srisa-an, and 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.

[17] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and 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.

[18] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[19] 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*, 2016, pp. 689–703.

[20] S. C. Misra and 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.

[21] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel SGX," in *27th USENIX Security Symposium*, 2018, pp. 1213–1227.

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

[23] SANS, "Glossary of security terms," 2019, https://www.sans.org/security-resources/glossary-of-terms/.

[24] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and 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.

[25] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and 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.

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

[27] llvm-admin team, "The LLVM Compiler Infrastructure," 2019, https://llvm.org/.

[28] The Clang Team, "LibTooling," 2019, https://clang.llvm.org/docs/LibTooling.html.

[29] Intel, "SGX sample code," 2019, https://github.com/intel/linux-sgx/tree/master/SampleCode.

[30] Google, "Google C++ style guide," 2019, https://google.github.io/styleguide/cppguide.html.

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

[32] Microsoft, "Naming guidelines," 2019, https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines.

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

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

[35] Google, "word2vec," 2019, https://code.google.com/archive/p/word2vec/.

[36] P. K. Sahoo, S. Soltani, and A. K. Wong, "A survey of thresholding techniques," *Computer vision, graphics, and image processing*, vol. 41, no. 2, pp. 233–260, 1988.

[37] The Clang Team, "Attributes in clang," 2018, https://clang.llvm.org/docs/AttributeReference.html.

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

[39] Google, "word2vec-GoogleNews-vectors," 2019, https://github.com/mmihaltz/word2vec-GoogleNews-vectors.

[40] Y. Liu, K. An, and E. Tilevich, "RT-Trust: Automated refactoring for different trusted execution environments under real-time constraints," *Journal of Computer Languages*, vol. 56, p. 100939, 2020.

[41] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "Sgxelide: enabling enclave code secrecy via self-modification," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2018, pp. 75–86.

[42] "GPS/GNSS Tracker for STM32Primer2," 2019, https://github.com/nemuisan/STM32Primer2_GPS_Tracker.

[43] "PAM module," 2019, https://github.com/tiwe-de/libpam-pwdfile.

[44] "su-exec," 2019, https://github.com/ncopa/su-exec.

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

[46] "Spritz Library For Arduino," 2019, https://github.com/abderraouf-adjal/ArduinoSpritzCipher.

[47] "libomron," 2019, https://github.com/openyou/libomron.

[48] "Ssniper Social Security Scanner for Linux," 2019, https://github.com/racooper/ssniper.

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

[50] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and 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.

[51] X. Xiao, X. Wang, Z. Cao, H. Wang, and 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.

[52] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing tcb complexity for security-sensitive applications: Three case studies," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4. ACM, 2006, pp. 161–174.

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