Methodologies, Techniques, and Tools for Understanding and Managing Sensitive Program Information

Yin Liu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Eli Tilevich, Chair
Shigeru Chiba
Muhammad Ali Gulzar
Na Meng
Francisco Servant

May 11, 2021
Blacksburg, Virginia

Keywords: Software Engineering, Program Analysis & Transformation, Program Comprehension, Trusted Execution Environment, Middleware

Methodologies, Techniques, and Tools for Understanding and Managing Sensitive Program Information

Yin Liu

(ABSTRACT)

Exfiltrating or tampering with certain business logic, algorithms, and data can harm the security and privacy of both organizations and end users. Collectively referred to as *sensitive program information (SPI)*, these building blocks are part and parcel of modern software systems in domains ranging from enterprise applications to cyberphysical setups. Hence, protecting SPI has become one of the most salient challenges of modern software development. However, several fundamental obstacles stand on the way of effective SPI protection: (1) understanding and locating the SPI for any realistically sized codebase by hand is hard; (2) manually isolating SPI to protect it is burdensome and error-prone; (3) if SPI is passed across distributed components within and across devices, it becomes vulnerable to security and privacy attacks. To address these problems, this dissertation research innovates in the realm of automated program analysis, code transformation, and novel programming abstractions to improve the state of the art in SPI protection. Specifically, this dissertation comprises three interrelated research thrusts that: (1) design and develop program analysis and programming support for inferring the usage semantics of program constructs, with the goal of helping developers understand and identify SPI; (2) provide powerful programming abstractions and tools that transform code automatically, with the goal of helping developers effectively isolate SPI from the rest of the codebase; (3) provide programming mechanism for distributed managed execution environments that hides SPI, with the goal of enabling components to exchange SPI safely and securely. The novel methodologies, techniques, and

software tools, supported by programming abstractions, automated program analysis, and code transformation of this dissertation research lay the groundwork for establishing a secure, understandable, and efficient foundation for protecting SPI.

This dissertation is based on 4 conference papers, presented at TrustCom'20, GPCE'20, GPCE'18, and ManLang'17, as well as 1 journal paper, published in Journal of Computer Languages (COLA).

Methodologies, Techniques, and Tools for Understanding and Managing Sensitive Program Information

Yin Liu

(GENERAL AUDIENCE ABSTRACT)

Some portions of a computer program can be sensitive, referred to as *sensitive program information (SPI)*. By compromising SPI, attackers can hurt user security/privacy. It is hard for developers to identify and protect SPI, particularly for large programs. This dissertation introduces novel methodologies, techniques, and software tools that facilitate software developments tasks concerned with locating and protecting SPI.

Dedication

For my family. Thanks for always being there for me.

Acknowledgments

I want to express my gratitude and appreciation for the following people who have helped me during my Ph.D. Journey:[1]

My advisor Dr. Eli Tilevich, for his mentoring, from which I benefit tremendously. I appreciate that he always spends lots of time on reviewing my paper, giving me constructive comments, helping me revise the paper from top to bottom as many times as it takes to achieve the top quality. I appreciate that he never hesitates to help me as much as possible when I come for help. I appreciate that he always guides me to find correct research directions when I discuss research topics with him. I appreciate that he always respects me, patiently guiding me to my best.

My committee members, Drs. Shigeru Chiba, Muhammad Ali Gulzar, Na Meng, and Francisco Servant, for their timely feedback and helpful suggestions. I appreciate their valuable insights that guide me to improve this dissertation's overall quality and further study relevant research topics.

My Software Innovations Lab mates and all CS@VT friends, for their every help in my research and daily life.

My family, for their continuous encouragement and support.

[1]This material is based upon work supported by the National Science Foundation (NSF) under Grants No. 1650540 and 1717065. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"I really think that if we change our own approach and thinking about what we have available to us, that is what will unlock our ability to truly excel in security." — *Greg York*

1.1 Problem Statement

Modern software systems commonly incorporate sensitive business logic, algorithms, and data, whose exfiltration or tampering would harm end users or entire organizations. These software building blocks are collectively referred to as sensitive program information (SPI). The leakage of sensitive data can compromise a user's security and privacy. For example, having stolen a user's password during an online banking session, an attacker can then log in to the victim's account and illicitly transfer funds. The exfiltration of sensitive business logic could weaken a company's competitiveness. For example, having reverse-engineered the business logic of a popular recommendation engine, a competitor can quickly spin off their own software product. Tampering with sensitive algorithms could perturb or even crash the system. For example, by tampering with a drone's navigation algorithm, an adversary can misroute the drone. The harmful consequences of such potential leakage, exfiltration, and tampering attacks call for effective mechanisms that can protect SPI. Although *SPI protection* is an established area in security research, the time is ripe for software engineering researchers to stand back and rethink, from the developer's perspective, what the most salient

issues of SPI protection are and which solutions we can contribute to address these issues.

From the developer's perspective, SPI comprises program variables that store data as well as functions that implement business logic and algorithms. Thus, protecting SPI translates into development tasks that ensure the integrity and confidentiality of sensitive variables and functions. Consider Alice, a software developer, tasked with protecting sensitive variables and functions in an existing system. To that end, she needs effective practical solutions that would provide definitive answers to the following three questions:

(1) How to identify variables that store SPI?

(2) How to protect these variables and functions operating on them?

(3) How to enable systems to securely exchange SPI?

Motivating Scenarios:

Scenario I. Assume that Alice aims to identify which of its parts can impact the drone's geolocation. To that end, Alice could start by searching for variables whose names bear similarity to the word "GPS." However, the actual names for geolocation variables often differ significantly from the word "GPS". A simple search of GitHub reveals the following names for GPS variables and their frequencies: "gps" – 16M cases, "geo" – 43M cases, "geolocation" – 7M cases, and "latitude" – 52M cases. Even worse, an undisciplined developer may name a geolocation variable with a single letter "g" or even an unrelated name "abc." Hence, relying on string matching alone would be quite inadequate in identifying all potential names of geolocation variables.

To address this problem, we present a new program analysis that infers a variable's usage semantics from its textual and context information (e.g., symbolic name, type, scope, information flow). To support this analysis, we introduce VarSem, a domain-specific language, in which a variable's semantic category is expressed as a set of declarative rules. Specifically, by using VarSem, Alice would be able to define rules that identify those variables

that are used in implementing geolocation-related functionalities. Then, these Alice-defined high-level rules would be translated into low-level natural language processing and data flow analyses, which calculate the likelihoods of each program variable belonging to the functionality of interest. That is, the calculated likelihoods would indicate whether a variable is used by some geolocation-related functionality, a piece of information that can help Alice identify the relevant sensitive variables.

Scenario II. Assume that Alice somehow has succeeded in correctly identifying the geolocation variables. Now, she would need to place these variables and their dependent functions into a secure execution environment that protects sensitive variables and functions from being compromised. Without adequate development tools, this task can be quite burdensome and error-prone.

To address this problem, we present RT-Trust that can help Alice to: 1) isolate the geolocation-related variables and functions; 2) redesign/redirect invocations of the dependent functions to communicate with the isolated constructs; 3) verify that the redesigned system continues to meet the original requirements (e.g., real-time constraints). Further, if Alice happens to misplace some functions into the secure execution environment, we present TEE-DRUP to identify and migrate these misplaced functions out.

Scenario III. Assume that Alice has succeeded in correctly placing all the relevant sensitive geolocation variables and functions into the secure execution environment. Now, she may need to make it possible to securely exchange the geolocation information across multiple apps running on the mobile platform that controls the drone. If an untrustworthy app receives or accesses raw geolocations, *data leakage* would occur. In fact, inter-component communication of modern mobile platforms (e.g., Android and iOS) remains vulnerable to data leakage, commonly exploited by security/privacy attacks that have been documented both in the research literature [36, 37, 43, 50, 55, 112, 170] and in vulnerability reporting

repositories (e.g., CVE [10]) [6, 8, 34, 120]. Hence, to complete her task, Alice has to protect the sensitive geolocation data from these attacks, an unmanageable task for anyone without specialized experience in designing secure systems.

To address this problem, we present ObEx, a programming mechanism for hiding sensitive data, that enables developers to effectively protect SPI by supporting predefined statistical queries on hidden data. Specifically, Alice can place the sensitive geolocation information into an ObEx object, configuring it to support the required set of query interfaces and expiration policies. When transferred from one host to another, the ObEx object keeps its sensitive data hidden, so it cannot be directly read or written. However, clients can execute the Alice-configured queries against the object. All of the ObEx object's sensitive data is reliably destroyed, once the specified lifetime expires, thus further preventing data leakage.

1.2 Research Contributions and Applications

The overriding goal of this dissertation research is to establish a secure, understandable, and efficient foundation for understanding and managing SPI. Specifically, this dissertation contributes to this goal in three ways:

(1) Design and develop program analysis and programming supports for inferring the usage semantics of program constructs, with the goal of helping developers understand and identify SPI (i.e., variable semantic analysis and VarSem);

(2) Provide powerful programming abstractions and tools that transform code automatically, with the goal of helping developers effectively isolate SPI from the rest of the codebase (i.e., TEE-insourcing, TEE-DRUP, and RT-Trust);

(3) Provide programming mechanism for distributed managed execution environments that

hides SPI, with the goal of enabling components to exchange SPI securely (i.e., ObEx).

Overall, we created novel concepts in the areas of program analysis and refactoring (e.g., variable semantic analysis, TEE-insourcing) and demonstrated their feasibility and utility by implementing new practical software tools (e.g., VarSem, RT-Trust, TEE-DRUP, ObEx). The former introduces insights that can guide the design of new specialized analysis and refactoring techniques. The latter can benefit software developers performing different roles. *For maintenance programmers*, VarSem can help in identifying variables with a specific usage in a large codebase; *for security analysts*, RT-Trust and TEE-DRUP can help in isolating the sensitive functions into TEE without incurring unnecessary performance costs, while ObEx can safely store and transfer sensitive data, thus avoiding leakages; *for real-time system developers*, RT-Trust can reduce the manual effort required to adapt systems for trusted execution under real-time constraints. Furthermore, by applying our methodologies, techniques, and tools to their projects, *software development organizations* can improve the security and privacy of their products, a critically important objective for all software-dependent environments.

1.3 Structure

The remainder of this dissertation is organized as follows. Chapter 2 introduces the technical background and discusses the related work of this research. Chapter 3 explains the main threat models and assumptions. Chapter 4 describes our approach to inferring usage semantics of program variables. Chapter 5 and Chapter 6 describe our approaches that isolate sensitive data and functions in TEE and migrate non-sensitive ones out, respectively. Chapter 7 describes our approach for secure data exchange. Chapter 8 and Chapter 9 present future work directions and conclusions, respectively.

Chapter 2

Background and Related Work

In this chapter, we first introduce definitions and technical background required to understand our contributions. We then discuss related work.

2.1 Definitions and Technical Background

Next we describe relevant definitions and the main technologies that power our approaches.

2.1.1 Definitions

Sensitive Program Information (SPI): The SPI can include business logic, algorithms, and data, if they happen to contain sensitive information whose exfiltration or tampering would harm end users or entire organizations. "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[1] refers to as "security terms" [139]. Specifically, *Sensitive* data (or variables) store security-related information or are referenced in security-related operations. *Sensitive* code (or functions) operates on *sensitive* data.

Textual & Context variable information: The properties of any program variable divide

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

into intrinsic (e.g., symbolic name, type, and scope) and extrinsic (data and control flow). *Textual information* refers to a variable's symbolic name, function name, type's developer-defined name (if applicable), and file path; *context information* refers to all its other properties (e.g., data type, data/control flow). Note that, the type name and the data type refer to different properties. While the former describes information that is textual, and the latter one is contextual. For example, consider a variable's type: `struct type_name`. The type name is "type_name" and the data type is "struct."

Personally identifiable information (PII)[140]: PII encompasses all information that would harm a person's security or privacy if exfiltrated. Typical PII examples are social security numbers, credit/debit card numbers, and healthcare-related data.

Real-Time Constraints: In general, real-time constraints [85] are the restrictions on the timing of events that should be satisfied by a real-time system; these restrictions can be classified into time deadlines and periodicity limits [105]. The former restricts the deadline by which a particular task must complete its execution. The latter restricts how often a given event should be triggered. For example, given the periodicity limit of 50ms and the time deadline of 20ms, a drone task must obtain its GPS location within 20ms for each 50ms period.

In our case, due to the memory limitation of the TEE, the event's memory consumption is another constraint. As we mentioned in Section 2.1.2, the TEE should maintain a small footprint by occupying limited space in memory. Also, if the TEE solution applies eMMC RPMB [134] as trusted storage only, the memory consumption is limited by the size of the RPMB partition, due to the persistent objects being stored in the RPMB.

As determined by how strict the timeliness requirements are, real-time constraints are categorized into hard and soft. The former constraints must be satisfied while the latter can be

tolerated with associated ranges. For example, a drone's motor/flight surface control must respond on time (hard constraint), while its navigation according to waypoints is expected to be resilient to deviations caused by GPS signal being temporarily lost or even wind gusts (soft constraint).

2.1.2 Trusted Execution Environment (TEE)

TEE [65] offers a standardized hardware solution that protects SPI from being compromised. First, TEE isolates a secure area of the CPU (i.e., the secure world for trusted applications) from the normal area (i.e., the normal world for common applications).[2]. That is, the secure world possesses a separate computing unit and an independent OS that prevents unauthorized external peripherals from directly executing the trusted tasks. In addition, TEE provides trusted storage that can only be accessed via the provided API to securely persist data. Finally, TEE offers an API to the secure communication channel, as the only avenue for external entities to communicate with the secure world.

OP-TEE[128]: Following the Global Platform Specifications of TEE, OP-TEE provides a hardware isolation mechanism that primarily relies on the ARM TrustZone, with three essential features: 1) it isolates the Trusted OS from the Rich OS (e.g., Linux) to protect the executions of Trusted Applications (TAs) via underlying hardware support; 2) it requires reasonable space to reside in the on-chip memory; 3) it can be easily pluggable to various architectures and hardware.

SGX[47]: Another implementation of TEE is Intel's Software Guard Extensions (SGX). It protects computation integrity and confidentiality by extending the Intel architecture. In

[2]The *normal* and *secure* world are the terms commonly used in the TEE realm. That is, if the code runs in the secure world, it is considered "trusted" (i.e., under protection); if it runs in the normal world, then it is considered "untrusted" (i.e., without protection and may be compromised).

the same way as OP-TEE, SGX requires that developers divide the original code into two parts: regular and trusted. The former runs inside of *the enclave*, a protected area that isolates the execution resources from the outside environment (kernel, hypervisor, etc.), in which the latter runs. Furthermore, the regular components can only access the enclave via special CPU instructions. Hence, if run or loaded inside the enclave, the application's SPI becomes invulnerable to attacks perpetrated from compromised outside environments.

2.1.3 Programming Languages and Analysis

Scala, a modern programming language, combines object-oriented and functional programming features [83]. Scala serves as the host language for VarSem, defined as an embedded DSL. We choose Scala because of its flexible syntax (e.g., optional parentheses), its support for defining new keywords and custom syntax, and functional programming features (e.g., support for higher-order functions). It is these features of Scala that make it possible for VarSem's library to be easily extended with new analysis techniques. Finally, as a JVM-based language, Scala runs on multiple platforms, making VarSem platform-independent.

Object Encapsulation: One of the fundamental concepts of object-oriented programming (OOP) is encapsulation, which hides the sensitive data and behavior from object clients. Moreover, Java provides access modifiers to ensure data privacy. By applying the keyword `private` to a field, programmers expect the field not to be accessible from outside of its declaring class. The protection afforded by Java access modifiers can be bypassed by using reflection. With the right permission, an attacker can use reflection to directly access and modify `private` fields and invoke `private` methods. To help prevent this attack, the security manager mechanism [70] has been added to Java, but its effectiveness depends on all components being properly configured, a requirement that can be hard to fulfill in complex distributed systems. Proper configuration and deployment practices can prevent these at-

tacks, but they require a universal adherence. In addition, to customize a security manager, developers have to overcome a steep learning curve. ObEx provides a novel encapsulation mechanism to protect sensitive data inside an object.

Object Life Cycle: When a programming object containing sensitive data goes out of scope, it becomes available for garbage collection. The actual collection time and scope are, however, entirely the prerogative of the collection policy in place. In some cases, however, waiting for the garbage collector to clear sensitive data may be insufficient. Instead, the sensitive data may need to be reliably cleared after reaching a certain threshold, as defined by the specified per-object access policy. ObEx provides such a mechanism that can enforce objects' life cycle to clear the sensitive data.

Natural language processing (NLP) approaches have been widely applied to identify program's sensitive textual information (e.g., comments, descriptions) [86, 124, 178]. VarSem relies on NLP to provide a novel textual analysis that is more powerful than that of a regular expression search. TEE-DRUP's NLP-based sensitivity analysis designates sensitive variables that should be protected in TEE.

Data-flow Analysis, a standard static program analysis technique, infers how values propagate through a program. Data-flow analysis has been applied to detect code vulnerabilities [41, 99, 138]. By employing the standard data-flow analysis, VarSem identifies variables' context information (e.g., information flows from user input to a variable).

LLVM [111] is a compiler-based program analysis infrastructure. LLVM features `libtooling tool` [151] and `AST- Matcher`[148] that analyze at the source code level. By means of a customized `libtooling tool`, TEE-DRUP and VarSem extracts the intrinsic properties of variables (e.g, name, type, and scope). RT-Trust and TEE-DRUP introduce a series of new `Pass`es that refactor SPI functions.

2.2 Related Work

This dissertation is related to several research area, including program semantics comprehension, code refactoring, and information leakage, we will discuss them in turn in this chapter.

2.2.1 Program Semantics Comprehension

To be able to effectively comprehend program semantics, a developer needs to have answers to the following questions: (1) what are the state-of-the-art comprehension strategies? (2) which software tools can be used to facilitate the comprehension process? (3) which comprehension technique can be used to detect the relevant program information? We discuss these questions in turn next.

Semantics Comprehension Strategies: Semantics comprehension strategies rely on dissimilar techniques and can be categorized into the following groups:

(a) data mining: Weimer et al. mine for code specifications beneficial for debugging a given codebases [164]. Høst et al. use data mining to extract rules from existing Java applications to identify unusual and unacceptable method names [84].

(b) static analysis: Mishne et al. apply static analysis-based semantic search over partial code snippets to understand API usage [121].

(c) ML/NLP: Allamanis et al. learn, via an n-gram language model, a code snippet's coding style to recommend identifier names and formatting conventions [21, 22]. Raychev et al. use structured support vector machines (SSVM) to predict identifiers names and type annotations in JavaScript projects [133]. Alon et al. apply deep learning to learn code embeddings, used to predict method names [25].

(d) others: Rice et al.'s algorithm detects whether a function call is passed correct parameters from an identifier's name [135]. Sridhara et al.'s algorithm generates documents for Java methods by selecting critical code statements and expressing them as natural language phrases [144]. Buse et al.'s descriptive model assigns human-annotated readability scores to source code features to measure code readability [38].

These solutions are intended for specific applications scenarios and rely on dissimilar theories and algorithms. They also focus on the source code's context information, without considering it in concert with textual information. In contrast, our semantic comprehension strategy — variable usage semantics analysis (VUSA) — considers both textual and context information to infer variable semantics.

Current Tools for Supporting Semantics Comprehension: Martin et al.'s PQL is a program query language for searching specific source code patterns [116]. Chen et al.'s VFQL is a program query language for expressing and searching for code defects through value flow graphs [40]. Urma et al. describe Wiggle, a querying system that uses Neo4j's Cypher language to search source code with user-specified textual or context properties [155]. Cohen et al. design a language for querying specific patterns in Java source code [46]. Some online tools and IDE plugins can locate given code patterns [27, 126]. However, none of these languages focus on inferring variable semantics. In contrast, our work — VarSem — reifies the novel *variable usage semantics analysis*, understanding variable semantics rather than that of the entire program.

Semantics Comprehension for Detecting SPI: 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 [86, 124]. UiRef solves the same problem while also resolving ambiguous

words [26]. ICONINTENT identifies sensitive UI widgets in Android apps by both resolving textual labels and classifying icons [168]. Different from these techniques, our work — TEE-DRUP — focuses on data as the origin of vulnerabilities by identifying sensitive variables.

2.2.2 Code Refactoring for Trusted Execution

To protect SPI, a developer needs to refactor the code of a program by going through three major steps: (1) partition the program into sensitive and non-sensitive parts; (2) transform the identified code and data into a TEE-based program for trusted execution, keeping in mind that the trusted computing base (TCB) must remain small; (3) ensure that the refactored code preserves the execution constraints (e.g., real-time constraints). Next, we discuss key existing work related to program partitioning, code transformation, and execution profiling / DSL for expressing and verifying constraints.

Partitioning Programs: J-Orchestra partitions the Java bytecode of a centralized application into a distributed application [152]. Given programmer annotations, Swift transforms a web application into a secure web application, in which the server-side Java part and the client-side JavaScript part interact with each other via HTTP [44]. ZØ compiles annotated C# code of a centralized application into a distributed multi-tier version to improve confidentiality and integrity, as directed by an automatically produced zero-knowledge proof of knowledge [60]. By enforcing a dynamic information flow control mechanism, Fission automatically and securely splits a JavaScript program into the client and server parts [80]. Pyxis automatically partitions database-backed applications into the application server and database parts [42]. Yang et al. optimize the code partitioning of mobile data stream applications [171].

Transforming Code and Data into TEE-based Programs:

(a) Moving code and data into TEE. PtrSplit partitions C-language systems, while automat-

ically tracking pointer bounds, thus enabling the automatic marshaling and unmarshaling of pointer parameters in RPC communication [106]. Senier et al. present a toolset that separates security protocols into several isolated partitions to fulfill security requirements [141]. Rubinov et al. leverage taint analysis to automatically partition Android applications for trusted execution [136]. TZSlicer automatically detects and slices away sensitive code fragments [173]. Lind et al.'s source-to-source transformation framework extracts subsets of C programs to take advantage of Intel SGX enclaves [103]. As compared with these works, our work — RT-Trust — not only supports the correct and automatic partitioning of legacy C code, but it also takes the real-time performance implications of the partitioning into account. By means of its profiling infrastructure and the *FPI* metric, RT-Trust predicts the degree to which a requested partitioning would decrease the system's real-time performance and also informs developers how to select between TEE implementations.

(b) Reduce trusted computing base (TCB). Singaravelu et al. identify the problem of large TCBs and how to reduce them in three real-world applications [143]. Lind et al. present an automated partition tool, Glamdring, that partitions a system by placing only the sensitive data and code in the TEE [103]. Qian et al. reduce the size of deployed binaries by developing RAZOR, an automatic code-debloating tool [132]. Rubinov et al. use FlowDroid's taint analysis to track developer-specified data, so only the relevant functions can be moved to TEE [136]. Our work, 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.

Execution Profiling and DSLs for Execution Constraints:

(a) Execution profiling tools. Several existing dynamic profiling tools, such as Pin tool [113], gperftools [77], and Gprof [78], ascertain program performance behavior. However, Pin and gperftools require that developers manually add profiling probes. Further, to profile program

in TEE, one would have to pre-deploy their dependent libraries, which may be incompatible with particular TEE implementations. Our work, RT-Trust, differs by automatically inserting profiling probes into the specified functions. Further, it estimates TEE-based execution characteristics without any pre-deployment.

(b) DSLs. Real Time Logic (RTL) formalizes real-time execution properties [95]. Subsequent DSLs for real-time systems include Hume that helps ensure that resource-limited, real-time systems meet execution constraints [81]. Flake et al. [58] add real-time constraints to the Object Constraint Language (OCL). Several efforts extend high-level programming languages to meet real-time execution requirements [35, 63, 93]. Our domain-specific annotations, Real-Time Trust Annotations (RTTA), can also be seen as a declarative DSL for real-time constraints, albeit to be maintained when the original real-time system is refactored to protect its SPI functionality.

2.2.3 Information Leakage

In this section, we discuss the prior work that aims to mitigate SPI leakage.

Data Privacy: Data privacy research divides between mathematical approaches and software engineering solutions. In mathematics, Dwork et al. [52] were first to put forward *differential privacy*, a mathematical solution that prevents attackers from maliciously discovering an individual's private information. As a protection mechanism, one can obfuscate the original dataset by generating random noise, while applying the mathematical frameworks of differential privacy to calibrate and measure the impact of the added noise on the statistical operations over the dataset [53]. Zhang et al. [177] proposed how an imperative language can be applied to facilitate the process of verifying differential privacy algorithms.

In software engineering, Gaboardi et al. [61] provided an information sharing interface that

enables users, without any differential privacy background, to conveniently generate privacy-preserving datasets that support statistical queries. Meanwhile, Liu et al. [104] presented a programming framework, Oblivm, a domain specific language that enables programmers to create cryptographic programs by using a custom compiler that generates code for secure computation. In addition, Kosba et al. [98] developed "Hawk", a blockchain-based contract system that compiles non-secure "Hawk" programs to those that guarantee the privacy of transactions by employing cryptography. Furthermore, Miller et al. [119] developed a tool that can generate secure protocols, enabling clients to security communicate with a non-trusted server.

Authentication and Access Control: Holford et al. [82] presented a self-defending object (SDO) that can authenticate users while invoking a method. An authentication token is passed as a parameter to the object's public methods to be able to examine whether the caller is permitted to invoke a given method. Meanwhile, Venelle et al. [157] provided a Mandatory Access Control (MAC) model for JVM that limits which methods can be invoked and which fields can be accessed. Android features secures data privacy via a permission system that defines which clients can access which resources [56, 57]. By contrast, our work, ObEx, disallows any direct read access for any clients, while enabling them to query sensitive data.

Self-Destruction: Vanish is a self-destruction system that periodically destroys expired access keys, so as to prevent all further access to the encrypted information [62]. Developing a similar concept, Xiong et al. [169] proposed an ABE-based secure document self-destruction (ADS) scheme for cloud-based sensitive documents. Meanwhile, SSDD offers a similar self-destruction scheme for electronic data [175]. However, Wolchok et al. [167] pointed out that attackers can compromise the Vanish approach by continuously crawling the storage to gain the keys. An external security framework improves the Vanish approach to defend against

sniffer attacks, albeit without being integrated with language runtime [176].

Language Protection Mechanisms: The Java standard API provides an internal protection mechanism for sensitive data via the `SignedObject` and the `SealedObject` mechanisms. The `SignedObject` stores a signature, thus protecting its serializable representation. Without the valid digital signature, the protected object cannot be extracted. The `SealedObject`, on the other hand, encrypts the original object, encapsulating the result with a cryptographic cipher. However, once the original object has been recovered, the sensitive data is no longer protected. Meanwhile, since the implementation of these mechanisms is solely in the Java layer, a native library could be used to intercept and compromise the security mechanisms provided by these objects.

Privacy-Preserving Database: Databases have been used ubiquitously in systems operating in environments that range from cloud servers to mobile devices. The state-of-the-art research on preserving database privacy can be categorized into two types:

By executing SQL queries on encrypted data, a) CryptDB defends against two types of information leakage attacks: database administrators inadvertently reveal the data, and adversaries compromise the database server or the database-backed applications [130]; b) Arx provides a functionally rich database with the AES's security level [129]; c) MONOMI enables high-performance complex queries on encrypted data [154]. Besides, some industry databases and tools apply similar query processes and provide privacy-preserving features [71, 117].

By introducing trusted hardware, Cipherbase [28], TrustDatabase [146], and StealthDB [158] place the database's core functions and sensitive operations into a secure area that is hard to compromise, protecting the data from information leakage attacks.

As compared with these prior works, our work — ObEx — relies neither on encryption nor on trusted hardware. The runtime reliably destroys the sensitive data, as prescribed

by a declarative access policy. The policies specify the lifetime of sensitive data, the types of queries allowed, and the number of queries permitted. In the meantime, ObEx manages sensitive data entirely in the runtime system, thus rendering the data invisible. Hence, ObEx can improve data privacy in all managed execution environments, with IoT and mobile applications being particularly promising as an application area. To further enhance the security of ObEx, we plan to experiment with placing its runtime into TEE as a future research direction.

Chapter 3

Threat Models

Much of this dissertation's technical contribution is concerned with improving security and privacy. In this research field, it is customary to define *a threat model*, specific scenarios that describe how adversaries can act against a system; to improve the system's security and privacy, one has to prevent or at least hinder these adversarial actions. This dissertation research comprises three major pieces: (1) program analysis and support for inferring the usage semantics of program variables (i.e., VarSem), (2) development tools and abstractions for isolating the sensitive data and functions (i.e., RT-Trust and TEE-DRUP), and (3) programming mechanism for secure data exchange (i.e., ObEx). Next, we discuss the attacker behaviors and assumptions, as they pertain to these three pieces.

3.1 Attacker Behaviors

(1) What attackers are expected to be unable to do:

a) We assume attackers are unable to modify the source code or intermediate representations, so as to mislead program analysis and transformation. Specifically, our approaches'(i.e., Varsem, TEE-DRUP, and RT-Trust) program analysis and transformation processes cannot be compromised by attackers. *b) We assume attackers cannot compromise the execution environment.* Specifically, for TEE-DRUP and RT-Trust, attackers cannot compromise the secure world of TEE. For ObEx, attackers cannot compromised the JVM.

(2) What attackers are expected to do:

Attackers may attempt to compromise the SPI in a given system. In fact, the possibility of information leakage sharply rises by the vulnerable functions [1, 2, 7], especially the functions processing sensitive data. For example, by compromising the data transmitting process, attackers maliciously obtain the current GPS locations [5]. In addition, arbitrarily exposing sensitive functions for interaction with external actors can be illegally exploited, which causes file deletion [4] or credential disclosure [3].

VarSem helps identify the SPI, and RT-Trust and TEE-DRUP refactor the system to protect the SPI. By placing only the true SPI code in TEE, they prevent attackers from compromising the SPI in the refactored systems. We will discuss these solutions in Chapters 4 5 and 6, respectively.

Further, attackers may attempt to compromise data exchange processes in order to leak sensitive information. For example, due to the man-in-the-middle attack, the Eview EV-07S GPS Tracker exposes lots of sensitive data, such as the current GPS location and IMEI numbers, when transmitting data over the Internet[1]. In the meantime, users' location is disclosed on the website because the Sleipnir Mobile application misapplies Geolocation API and sends the sensitive data without gaining user permission[2]. What is worse is that, whether shared legitimately or inappropriately, sensitive data can be stored persistently. Attackers can then use that sensitive data to perpetrate a variety of subsequent privacy exploits. In addition, attackers may attempt to compromise Java in-memory objects that contain sensitive data. In fact, malicious accesses to Java objects have threatened a large number of applications [3] [4] [5]. The built-in Java language protection mechanisms, such as

[1]CVE-2017-5239 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5239
[2]CVE-2014-0806 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0806
[3]CVE-2009-1084 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1084
[4]CVE-2009-2747 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2747
[5]CVE-2012-0393 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0393

object encapsulation and garbage collection can be insufficient to defend against particularly elaborate attacks, particularly in distributed environments.

ObEx blocks read and write access to sensitive data, while enabling developers to use the statistical properties of this data in their applications. We will discuss it in chapter 7.

3.2 Assumptions

1) We assume that most of the variables, functions, and files in the program are named meaningfully. 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 [72, 87, 118]. Regular code reviews often come up with suggestions how to rename identifiers to more meaningfully reflect their roles and usage scenarios [73].

2) We assume that the users of our techniques possess sufficient background about the architecture of the analyzed software systems. To be able to describe target variables, VarSem users are expected to possess some domain knowledge about how the functionality of interest is implemented. For example, to specify variables that store passwords, a VarSem user is expected to understand how password-based authentication works in the analyzed projects. For example, in a certain "login" function, a password candidate, entered by an end user, is compared with the known password, retrieved from some storage. To be able to annotate SPI functions, RT-Trust and TEE-DRUP users are expected to be able to differentiate between sensitive and non-sensitive variables/functions. To be able to protect sensitive data inside an ObEx object, ObEx users are expected to be able to define when and how clients should query the properties of sensitive data objects (i.e., how long the data remains accessible, how many times its properties can be queried, which data query methods apply, etc.),

Chapter 4

Inferring Usage Semantics of Program Variables

In a computer program, variables are introduced for specific implementation purposes. We refer to such purposes as *variable usage semantics*. Understanding variable usage semantics is required for the majority of maintenance and evolution tasks, ranging from program comprehension to code audits [64, 101]. To that end, both the textual (e.g., naming scheme) and context (e.g., data flow) information of a variable should be examined, as it is their confluence that uncovers the variable's intent and responsibility [101].

Consider the task of identifying the variables that store user-entered passwords. This task is required for inspecting how passwords are protected in a system. As is shown in Figure 4.1, a security engineer, would need to ❶ search for variables whose names bear similarity to the word "password" (e.g., "pwd", "passwd", and "pass_word"), ❷ search for variables whose type can store password information (i.e., string), and ❸ search for variables whose context information matches certain usage patterns (e.g., information flows from user input to a variable). These search rules can be executed either independently or as a logical chain.

For task ❶, programmers typically use a source code search facility, such as the Unix grep. For example, issuing the command `grep "pass*word" *.c` on the Unix shell would return all the textual matches with the prefix "pass" and the suffix "word" contained in the C source files located in the current directory. However, this method is quite brittle: the actual names

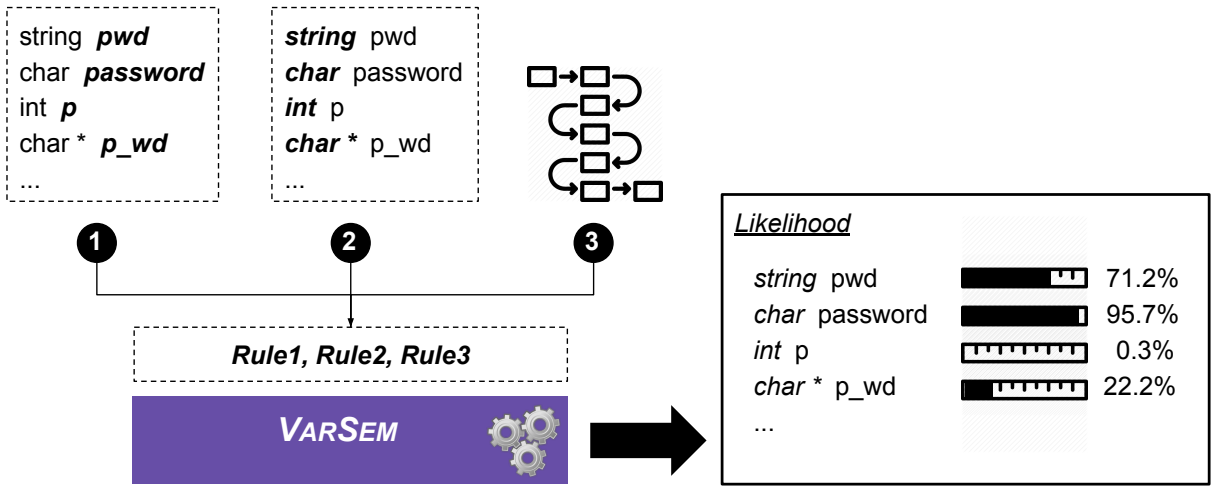

Figure 4.1: Identify variables storing user-entered passwords

for password variables often differ greatly from the word "password". A simple search of GitHub reveals the following names for password variables and their frequencies: "passwd" – 11M cases, "p_wd" – 9,258 cases, "pass_wd" – 280 cases, and "p_w_d" – 164 cases. Even worse, a lazy developer may name a password variable with a single letter "p" or an irrelevant string "abc." Hence, string matching would be quite inadequate in its ability to identify all potential password variable names. Besides, not all matches would be against variables, as some of them will appear in comments, annotations, and class/function names, thereby requiring additional manual efforts to filter out.

For tasks ❷ and ❸, manually inspecting the variables' types and information flows would be feasible for small projects, but quite unrealistic for large projects. Compiler-based analysis tools (e.g., LLVM ASTMatcher [148]) and existing program analysis frameworks [107, 138, 162] can be used to search for variable types and information flows automatically. However, to apply these tools correctly requires that programmers build sufficient expertise in compilers and program analysis. It would be unrealistic to expect that level of expertise in esoteric topics from all application programmers or security engineers.

The latest developments in machine learning (ML), deep learning (DL), and natural language processing (NLP) provide new tools for analyzing variable usage semantics. However, applying these state-of-the-art approaches correctly is quite challenging due to the following reasons: **(a)** existing applications of these tools are quite dissimilar, including recommending identifier names [21, 22, 24, 25], summarizing code [23, 161], de-obfuscating code [156], completing code [127], and debugging method names [84]. In other words, these approaches apply dissimilar ML/DL/NLP models to fit their specific scenarios, so no single model or technique could be used out of the box for variable usage semantics analysis; **(b)** the accuracy, precision, and recall of many of these techniques are insufficient for practical variable usage semantics analysis tasks. Consider recommending descriptive identifier names as an example: the state-of-the-art approach code2vec [25] achieves an imperfect precision (63.1%) and recall (54.4%). Even worse, Jiang et al. [96] show that in more realistic settings, code2vec's performance decreases even further; **(c)** the targets of these techniques are usually code segments rather than individual variables. That is, the contained variables' textual and context information are treated as tokens and attributes of the entire code snippet. However, variables are distinct program constructs. Not only does a variable store information, but it also can interact with other program constructs (e.g., class fields, function parameters) and be part of every program context (e.g., control and data flow). Understanding what variables are used for (i.e., their usage semantics) has not been pursued previously as a distinct program analysis problem.

What if one could write simple statements that would execute tasks ❶❷❸ by correctly employing the required state-of-the-art analysis techniques and tools as well as leveraging the available domain knowledge (e.g., software/security engineers)? In this chapter, we present *variable usage semantics analysis*, a new program analysis that identifies variable usage semantics, and VarSem, a domain-specific language that reifies this analysis. As shown

in Figure 4.1, one can declaratively specify the rules to execute tasks ❶ (Rule1: compare variables' symbolic names to the word "password"), ❷ (Rule2: identify variables whose type is `string`), and ❸ (Rule3: identify variables whose values flow from user input). By executing the specified rules, VarSem assigns a statistical probability value to each program variable to designate how closely it matches the given rules (e.g., pwd 71.2%), thus assisting the user in understanding the variables' usage semantics.

To ensure satisfactory accuracy/precision/recall, VarSem programs express both the textual and context information of variables. VarSem provides built-in analyses: data-flow for the context information and a novel NLP-based analysis for the textual information. To cover potentially enormous analysis scenarios, VarSem's extensible architecture enables software/security engineers to contribute highly customizable rules, and integrate with other program analysis techniques.

The contribution of this work is three-fold:

(1) We introduce a new program analysis, *variable usage semantics analysis (VUSA)*, that infers a variable's usage semantics from its textual and context information.

(2) We reify VUSA as the VarSem DSL that offers: (a) an intuitive declarative programming model; (b) built-in analyses: an NLP-based and data-flow analyses, for textual and context information, respectively; (c) an extensible architecture for including custom rules/analyses.

(3) We evaluate VarSem's performance in inferring personally identifiable information (PII) variables in eight open-sourced projects. VarSem infers variable semantics with satisfying *accuracy* (>= 80% in 13 out of 16 scenarios), *precision* (>= 80% in 13 out of 16 scenarios), and passable *recall* (> 60% in 8 out of 16 scenarios).

4.1 Design Philosophy and Overview

We first introduce the key idea behind VarSem and explain its high-level design. Then, we discuss the technical background required to understand our contribution. Finally, we revisit the motivating example above by applying VarSem.

4.1.1 Variable Usage Semantics Analysis

Our basic idea of *variable usage semantics analysis* is inspired by philosopher Ludwig Wittgenstein's Theory of Natural Language Construction: "the meaning of a word is its use in the language [166]." That is, the meaning of a word is determined not only by its textual representation, but also by the total set of usages of this word in the language. As a result, a word's meaning is not immutable: once its new usages appear and become acceptable, its meaning will gradually change as well. Hence, any dictionary-provided definition can only approximate the actual meaning of a word [84].

Inspired by Wittgenstein's theory, we treat program variables as words in a natural language, so the semantics of variable usage can be interpreted analogously to the meaning of words. Thus, to infer a variable's usage semantics, our approach is to consider both the variable's symbolic name (i.e., textual information) and how it is used in the program (i.e., context information).

A typical practical application of *variable usage semantics analysis* is determining the likelihood of variables belonging to a given *semantic category* (e.g., "password", "user information", "phone number"). Because of the intrinsic ambiguity of determining such likelihood, our approach relies on statistical probability to report the inferred results. Our approach defines a semantic category as a set of rules against which each program variable is evaluated.

When matched, a rule increases the likelihood of a variable belonging to the category specified by the whole rulebase. This approach is driven by the well-known insight that the results of data analysis depend on the analyst's domain knowledge (e.g., the analyst understands how password variables are used in a system) [156, 165]. The ability to specify a highly configurable rulebase, with both predefined rules and new customized rules, avails VarSem for the dissimilar needs of domain experts in need of inferring variable usage semantics.

4.1.2 Motivating Example Revisited

Recall the problem of identifying the variables that store user-entered passwords. Variables that belong to this semantic category match the following three rules: ❶ their symbolic names bear close similarity to the word "password", ❷ their type must be able to store textual data (i.e., `string`), and ❸ their value must have flown from some input. In VarSem, these rules can be expressed in a few lines of code as follows:

```
1 sem ("identifyPWD") {
2     $var.Name ~ "password"
3     $var.Type ~ STRING
4     $var.Value <~~ USER_INPUT
5 }
6 Result res = run ("identifyPWD")
7              on ("./src/*.c") threshold 0.8
```

The keyword **`sem`** creates a new semantic category, identified by its unique parenthesized name (**`sem`** `("indentifyPWD")` on line-1). This semantic category is defined by the three rules described above: ❶ (line-2), ❷ (line-3), and ❸ (line-4), respectively. The keyword **`$var`** represents a program variable, and it includes attributes (e.g., `Name`, `Type`, `Value`) that can be used to form rules. VarSem provides several built-in operators (e.g., `~`: match, `<~~`:

information flow), each of which returns a statistical probability, expressed as a percentage point. The informal semantics of Rule ❶ on line 2 is "compare a variable's symbolic name to the word 'password', returning their degree of similarity." Rule ❷ determines whether a given variable's type can hold a textual representation (e.g., `char *`, `char[]`, `const char *`, `string`). Rule ❸ determines whether an information flows from any user input to the given variable's value. Each program variable is bound to `var`, one at a time, with the rules executed, once their semantic category ("identifyPWD") is `run` on a given codebase (lines 6,7). These rules execute in any order, in sequence or in parallel. The optional `threshold` filter applies a weighted average formula to report only those variables whose likelihood of matching the rules are above the given threshold.

4.1.3 Required Language Features

The example above is simplified for ease of exposition. To become a truly versatile tool for inferring variable usage semantics, VarSem must provide the following features:

(1) Rules should be straightforward to reuse within and across semantic categories. It should be possible to make the execution of a rule conditional on the results of executing other rules.

To provide feature (1), a semantic category supports add/remove/update rule operations (4.2.2-3). Besides, the results of running a category can be examined, with the results used as conditional and control flow statements (4.2.3-2).

(2) It should be straightforward to redefine how rules and their analysis routines are bound to each other. Although each built-in rule has a default analysis routine, it should be possible to change these routines.

To provide feature (2), the `bind` keyword makes it possible to bind new analysis routines to existing rules (4.2.2-2).

(3) It should be possible for rules to have different levels of importance. When determining whether a variable belongs to a given semantic category, each rule can have a dissimilar impact. For example, how can the user specify that it is more important for a variable's name to be similar to the word "password" than to have some user input to flow into the variable's value?

To provide feature (3), a rule can be followed by the keyword `impact` and an integer value between 1 and 10 (4.2.2-2).

(4) VarSem should be extensible with both new rules and their analysis routines. Due to the large number and variety of analysis techniques and tools, it would be highly advantageous to be able to add them to VarSem or use them to update the analysis routine of existing rules.

To provide feature (4), VarSem can generate keywords/operators for new rules and function skeletons for corresponding analysis routines (4.2.4).

4.2 The VarSem Language

We start by presenting VarSem's high-level architecture. Then, we briefly introduce the major technologies that enable VarSem. Finally, we detail each major VarSem component.

4.2.1 Architecture

Figure 6.1 shows the three main components of VarSem: `Rules`, `Library`, and `Runtime`. `Rules` are user-defined search criteria that define a semantic category. `Library` provides a collection of program analysis routines (i.e., green circles), both standard and custom, that are bound to rules (i.e., red triangles) to reify them. The bindings can be default or user-

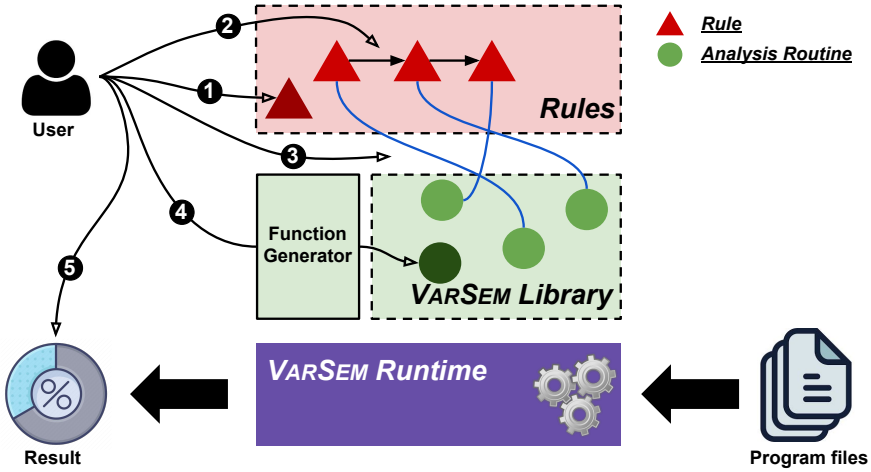

Figure 4.2: VarSem overview

defined. For example, a standard data-flow analysis method can be bound to the rule — "variable's value comes from user input." `Runtime` executes the rules in VarSem's semantic category by invoking the bound methods. It keeps track of each rule's execution, weighted-averaging the statistical probability of each program variable's belonging to the semantic category specified by the rulebase.

VarSem's programming model supports five inter-related development tasks: ❶ define a semantic category, ❷ add/remove/update rules in a semantic category, ❸ bind rules to analysis routines, ❹ create new rules and analysis routines, and ❺ apply a semantic category to a codebase, filtering and analyzing the results. Thus, by defining a semantic category with its associated rules (❶❷), VarSem can flexibly express a variable's combined textual and context information. VarSem's built-in rules and their analysis routines make use of a novel NLP-based analysis (for textual information) and classic data-flow techniques (for context information). By creating and binding rules and their analysis routines (❸❹), VarSem can be extended with new and improved analysis techniques and frameworks. Finally, VarSem's execution results can be analyzed and filtered as required to support various scenarios in inferring variable usage semantics (❺).

4.2.2 Syntax, Semantic Category, and Rules

The two key abstractions of VarSem are *a semantic category* and *rules*, explained in turn next.

(1) Semantic Category defines a category of variables with the same usage semantics. In VarSem, it is expressed as a group of rules (i.e., a rulebase) that describes a variable's intrinsic and extrinsic properties. A variable's membership in a semantic category is determined by the variable matching the category's rules. The keyword `sem` creates a new semantic category that comprises a unique name and a rulebase.

(2) Rule matches each program variable's intrinsic or extrinsic properties. A typical VarSem rule includes an operator and two operands. The *lhs* operand is an analyzed variable, while the *rhs* operand is a specific target (i.e., intrinsic or extrinsic targets). A rule can also include its impact modifier and custom analysis routine.

To describe a variable's intrinsic properties, a rule starts with the keyword **`$var`**, an intrinsic attribute (`Name`, `Type`, or `Scope`), the intrinsic operator (`~`), and the attribute's match-target. VarSem predefines several match-targets: string values for matching the attribute `Name`, `STRING`/`INT`/`BOOL`/`DOUBLE`/`COM- POSITE` for the attribute `TYPE`, and `LOCAL`/`GLOBAL` for the attribute `Scope`. `$var.Scope ~ LOCAL` expresses "match if the variable's scope is local."

To describe a variable's extrinsic properties, a rule starts with the keyword **`$var`**, the extrinsic attribute (`Value`), one of VarSem's extrinsic operators: data flow (`<~~` or `~~>`) or comparison (`<=>`), and one of the right-hand side (rhs) operands. The data flow operator's rhs operands can be one of `USER_INPUT`, `STABLE_STORAGE`, `WWW`, or a function name. The comparison operator's rhs operands can be either another variable's attribute (i.e., `Value`) or no operand. The comparison operator with no rhs operand (e.g., `$var.Value <=>`), expresses that any comparison with the variable's value should be matched.

To create rules that involve more than one variable, VarSem provides the `var_` operator. If more than two variables need to be differentiated, `var_` takes optional numeric parameters (e.g., `var_(1)`). To describe scenarios that use extrinsic properties, multiple rules can be combined. For example, the code snippet below describes one such scenario: one variable flows from some user input (line 1), another variable flows from some stable storage (line 2), and these two variables are compared (line 3).

```
1   $var.Value <~~ USER_INPUT &
2   $var_.Value <~~ STABLE_STORAGE &
3   $var.Value <=> $var_.Value
```

Because each matched rule can dissimilarly impact whether a variable belongs in a given semantic category, VarSem provides the **`impact`** keyword that takes an integer in the range [1, 10]. Impact levels control the calculation of weighted averages that determine which variables belong to the specified semantic category (see 4.2.3).

Finally, a specific analysis routine can be bound to a rule using the **`bind`** keyword. VarSem's standard library comes with a set of predefined analysis routines for core analysis techniques (e.g., `NLP_VAR_NAME_ANALYSIS`, `STRING_MATCH`, and `DATA_FLOW_ANALYSIS`). For example, the code snippet below is the rule ("match a variable's name to the word 'password"'), with the rule impacting the final statistical result to the degree of 7, and the analysis routine `NLP_VAR_NAME_ANALYSIS` performing the required analysis.

```
1   $var.Name ~ "password" impact 7
2             bind NLP_VAR_NAME_ANALYSIS
```

If **`impact`** is not specified, the default value of 10 is assigned. Without custom analysis routines, VarSem binds `NLP_VAR_NAME_ANALYSIS` for rules of textual information and `DATA_FLOW_ANALYSIS` for rules of context information.

(3) Supporting Reuse and Evolution of VarSem programs. Both semantics categories and individual rules can be systematically reused and evolved. In particular, rules can be inserted, removed, or updated in a semantic category, while semantic categories can be combined or intersected.

4.2.3 Reporting Output

(1) Obtaining results. The overloaded function `run` takes either a single rule or a semantic category as the parameter. The keyword `on` specifies the target codebase. The returned `Result` is a collection of key-value pairs, where the key points to a program variable and the value points to the likelihood of the variable belonging to the specified semantic category. The returned collections can be filtered to include only those variables whose likelihood exceeds a given threshold. The code snippet below runs the semantic category "identifyPWD" on the codebase located in "./src/*.c", and filters out all the variables whose likelihood value is less than 0.8.

```
1 Result res = run ("identifyPWD")
2               on ("./src/*.c") threshold 0.8
```

(2) Analyzing the results: The `Result` object provides the max, min, mean, and standard deviation operations, to be executed against the likelihood values. In addition, top/bottom percentages can be retrieved and filtered out. The code snippet below retrieves the maximal likelihood value from "res" (line 1), and then retrieves those variables whose likelihood value is in the top 1% (line2).

```
1 var max = res.max
2 res = res.top(0.1)
```

Conditional and control flow statements can operate on `Result`. The code snippet below has the following logic: if the results of executing the semantic category ("identifyPWD") are unsatisfactory (i.e., the max likelihood is less than 0.9), then run another semantic category ("identifyPWD_2").

```
1 Result res = run ("identifyPWD") on ("./src/*.c")
2 if (res.max < 0.9)
3     res = run ("identifyPWD_2") on ("./src/*.c")
```

Revisiting The Motivating Example Again Consider evolving the motivating example to return those variables whose likelihood values are in the top 1% and their mean value greater than 0.7. Further, we want to increase the importance of the variable's name. The code snippet accomplishes these changes.

```
1 Result res = run ("identifyPWD") on ("./src/*.c")
2 for (i <- 1 to 10 if res.top(0.1).mean > 0.7) {
3     update ("identifyPWD") {$var.Name ~ "password"}
4     to {$var.Name ~ "password" impact i}
5     res = run ("identifyPWD") on ("./src/*.c")
6 }
```

Please, note that even with an `impact` suffix, the updated rules are identified by their `$var.Name ~ "password"` prefix.

4.2.4 Meta-Programming Support for Creating new Operators and their Analysis Routines

Because variable usage can be defined in a variety of different ways, it would be impossible to provide a fixed set of rules and their analysis techniques to cover all possible scenarios. To

make it possible to easily create new rules and their analysis routines, VarSem provides meta-programming facilities. A new attribute can be created with the keyword **attr** followed by the attribute's name. A new operator can be created with the keyword **op** followed by the operator's symbols. VarSem supports the creation of binary operators only, so each new operator has to be accompanied with the declarations of its right-hand side and left-hand side operands by means of the **lhs** and **rhs** keywords, respectively. A new analysis routine can be created with the keyword **impl** followed by the analysis routine's unique name. The code snippet below shows how to add to VarSem a new attribute "Const"(line 1), a new operator"tainted" with its lhs "$var.Value" (i.e., variable's attribute), rhs "("source_f","sink_f")" (i.e., a string pair), and an analysis routine "TAINT" that can be bound to this rule (line 2). Given this meta-programming declaration, VarSem will generate a library that reifies the new attribute (usage example: `$var.Const`), as well as the new operator `tainted` and its analysis routine "TAINT" (usage example: `$var.Value tainted ("source_function","sink_function")bind "TAINT"`).

Note that, it would be impossible to generate a complete analysis routine, so VarSem generates skeletal code that is to be completed as necessary. For example, in the code snippet below, VarSem generates the analysis routine skeleton "TAINT", which uses the variable's attribute information and the string pair ("source_function", "sink_function") as the parameters. To complete the analysis routine, the skeletal body must be filled with the required logic.

```
1 attr "Const"
2 lhs $var.Value op "tainted"
3       rhs ("source_f","sink_f") impl "TAINT"
```

4.3 Executing VarSem Programs

We first discuss how VarSem executes rules in a given rulebase and then describe VarSem's built-in library.

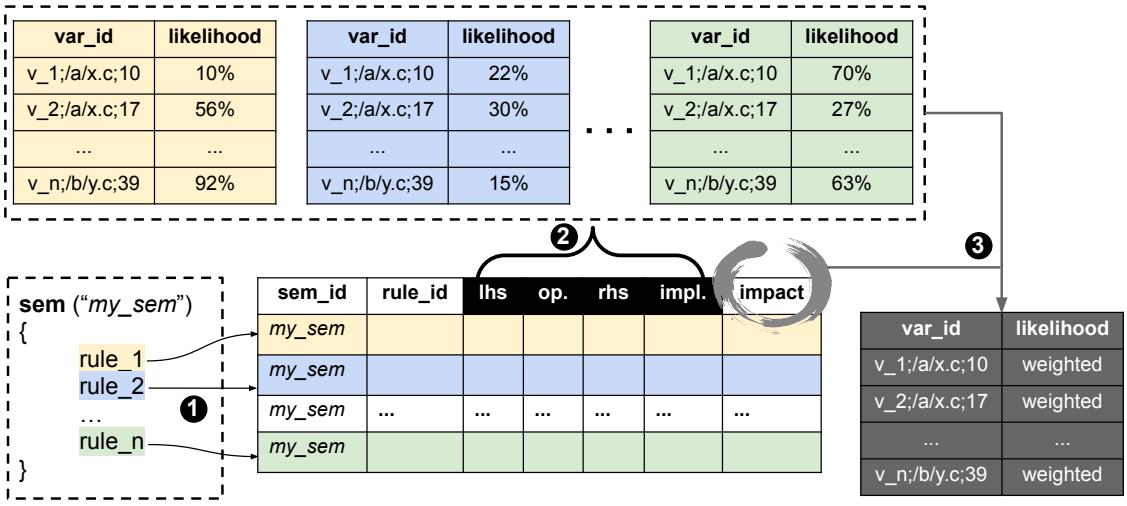

Figure 4.3: Executing VarSem rules

4.3.1 Executing Rules

Figure 4.3 details how VarSem executes the rules in a rulebase. Each included rule is processed in three steps:

❶ generate a rule's information vector: semantic category id, rule id, lhs, operator, rhs, bound analysis routine, and impact level. Specifically, the combination of semantic category id and rule id defines a rule's unique identity; the operator, lhs, rhs, and bound analysis routine determine how to execute the rule; the impact level determines how to weight the rule's results. As discussed in 4.2.2, if the impact level or analysis routine is omitted, VarSem assigns the default impact value of 10, and binds the rule to its built-in analysis routine (i.e., `NLP_VAR_NAME_ANALYSIS` for analyzing textual information ; `DATA_FLOW_ANALYSIS` for analyzing

context information).

❷ each rule in a rulebase is executed in the listing order (see 4.3.2). Each rule follows a strict syntax (i.e., lhs op rhs) and outputs either a percentage or a boolean value for likelihood. VarSem also follows a uniformed format when outputting the intermediate results: a unique id (a variable's name, file path, and line number) and a likelihood value (i.e., see the top of Figure 4.3). Recall that the likelihood value is the statistical probability of a program variable belonging to the specified semantic category.

❸ the likelihood values are weighted by their impact levels. These weighted results are returned as the final output (i.e., the grey table in Figure 4.3) as an array of mappings.

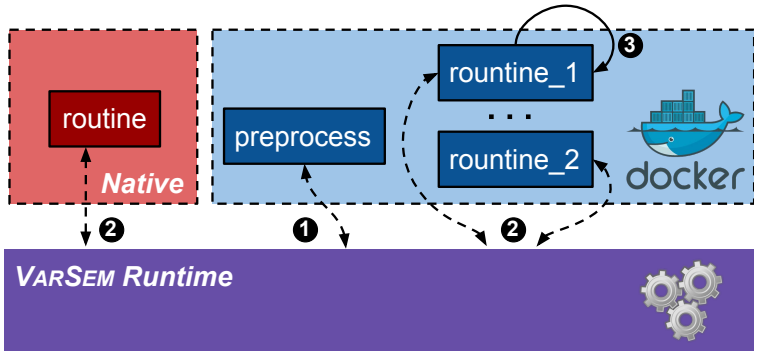

Figure 4.4: Executing Libraries

❶ preprocessing a codebase. Before applying any analysis routine, the runtime makes the target codebase readable by moving it to a shared folder. From the moved codebase, preprocessing extracts the variables' intrinsic properties (e.g., name, type, scope).

❷ invoking analysis routines. To invoke a native analysis routine, the runtime calls its entry point's Scala or Java function. To invoke a containerized routine, the runtime launches the corresponding container, so the entry point's function can be called via Docker APIs.

❸ adding containerized modules. New containerized routines can be added to the Docker, while existing routines can be updated through a configuration file.

4.3.2 Customizable Runtime

Because VarSem's runtime uses various program analysis techniques, it must be highly configurable to support their dissimilar execution environments. Hence, an advanced user can

add to VarSem not only custom analysis routines, but also their specialized execution environments. Although the VarSem's native runtime environment is Scala on the JVM, a new analysis routine may require Python support, as is common for machine learning-based techniques. VarSem supports custom execution environments by deploying them in a Docker container (Figure 4.4).

4.4 NLP-Powered Textual Analysis

VarSem includes built-in context and textual analysis routines. A stock data-flow analysis is used out-of-the-box for context information, albeit configured and invoked via declarative rules. In contrast, for VarSem's built-in textual information analysis, we created a novel technique, powered by NLP, that we discuss next.

(1) Rationale. When determining whether a variable belongs to a semantic category, its textual information needs to be compared with some target value to compute their similarity. Recall our motivating example of identifying variables that store user-entered passwords. The rule `$var.Name ~ "password"` matches a variable's symbolic name with the string "password." However, this match would be shallow and insufficient if other aspects of textual information, including type, enclosing unit, and file path, were not taken into account as explained next:

a) Variables with a specific usage tend to appear in certain enclosing units (e.g., functions) and source files. For example, the variable named "pwd" is more likely to store passwords if it is also referenced within a function named "login" whose implementation source code appears in the "authenticate.c" file. However, if a variable with the same symbolic name is referenced within the "unix_shell" function, its usage is likely be related to the implementation of the Unix pwd command (i.e., print working directory), an unrelated semantic usage category altogether.

b) If a variable's type is developer-defined (e.g., struct/class names), the specified type name can be indicative of the variable's usage. For example, many such variable types in our evaluation subjects (e.g., `struct feature_info_t` in the"Bio- metric Authentication" project) have type names that reveal their variables' usage semantics (6.4).

c) Adjacent words are semantically connected. That is, if an obscurely named identifier is adjacent to an obviously named identifier, they are semantically related. It is also likely that both of them are describable by the obvious name. For example, the obscurely named variable "pd" whose source file is in the path "a/b/c/password/x.c" should be more likely to store a password than if its source file path were "password/a/b/c/x.c". Although the word "password" appears in both file paths, its position is closer to the source file containing the variable in the first path.

(2) Processing Steps. VarSem uses NLP to compute the similarity between each encountered program variable's textual info and the target word ("password" in our motivating example). The similarity score then determines the likelihood of variables matching the specified usage semantics.

a) Pre-processing: To be used for analysis, the extracted variable's textual information is pre-processed: splitting/combining identifiers and removing redundancies.

Splitting/Concatenating Identifiers: Even if variables, types, functions, files, and directories are named in a meaningful way, their names can follow dissimilar naming conventions, such as delimiter-separated (e.g., the_pass_word) or camel case (e.g., thePassWord).

To be able to process such names irrespective of their naming conventions, their textual information is pre-processed into semantic sets by splitting and concatenating their constituent substrings. This pre-processing is demonstrated by example below for the names `a_b_c` and

`aBcDe`:

$$a\_b\_c \rightarrow [a, b, c], [ab, c], [a, bc], [abc] \tag{4.1}$$

$$aBcDe \rightarrow [a, bc, de], [abc, de], [a, bcde], [abcde] \tag{4.2}$$

More specifically, an identifier is first split into its constituent parts, sans delimiters, each of which is lower-cased. For example, both "the_pass_word" and "thePassWord" would become identical sets of "the", "pass", and "word." Then, the resulting parts are concatenated into semantic sets. For example, "the_pass_word" could become [the, pass, word], [thepass, word], [the, password], and [thepassword]. Similarly, the semantic sets can be generated for an identifier, separatable into multiple parts (e.g., it_is_a_real_pass_word as a rare case example). Finally, for each semantic set, its similarity score is computed, with the highest observed similarity score reported as the final result.

Removing Redundancies: Only certain identifier parts indicate their construct's usage semantics. For example, in "the_password", "password" is highly indicative, while "the" provides no useful information. Hence, dictionary-based removal gets rid of those parts that correspond to prepositions (e.g., in, on, at), pronouns, articles, and tense construction verbs (i.e., be, have, and their variants). For "the_password", "password" will be retained and "the" removed.

b) NLP Analysis: Algorithm-2's function `NLP_main` outputs variables and their statistical matching likelihood, given a collection of program variables (i.e., `variables`) and a target word (i.e., `target_word`). Each input variable is analyzed, and its identifiers are extracted from the pre-processed variable's name, type, function, and file path (lines 17,19,21,23). Next, function `get_similarity` computes the similarity score between each extracted identifier and the target word (lines 18,20,22,24). Each extracted identifier is broken into constituent

words, or combined to a single word. This process generates two representations as described above (i.e., `pass_word` → (1) `pass`, `word`; → (2) `password`). Each representation is tokenized (line 3). For each token, its similarity to the target word is computed (lines 5-6), with the similarities accumulated (line 7) and averaged (line 9). The maximum similarity is then selected (lines 10-12) and returned (line 14). An attenuation rate, λ, differentiates adjacent vs. nonadjacent semantic connections. The sum of the obtained similarities of identifiers is the variable's statistical matching likelihood (line 25).

c) An Example: Following the example in 4.1.2, consider how variables would be matched with word "password." Assume the variable collection contains variable `pass_word_info` with type `struct information`. The variable is accessed within function `login`, defined in "src/login.c". The name `pass_word_info` is pre-processed into four representations: [`pass, word, info`], [`pass, wordinfo`], [`password, info`], and [`passwordinfo`], while the type, function, and file path into [`information`], [`login`], and [`src, login`], respectively. Each of the tokens `[pass, word, info]` is matched with the target `password`, obtaining the similarity scores 0.5, 0.2, and 0.1, respectively. Hence, the resulting similarity scores becomes 0.27 (i.e., (0.5 + 0.2 + 0.1)/3). The procedure is applied to all representations as well as to those of the corresponding type, function, and file path, with the highest score passed to the next step. All similarity scores are scaled by λ (80% by default). That is, for `pass_word_info`'s file path array [`src, login`], the similarity of `src` is scaled by 80%. If the original similarity of `src` is 1, then it becomes 0.8 when scaled (i.e., the original value multiplies λ: $1 * 80\%$). Finally, the resulting similarity scores are summed and normalized to the range of 0 to 1 (i.e., a percentage value), which becomes the reported statistical matching likelihood.

Algorithm 1 VarSem's variable labeling.

Input : variables (i.e., a collection of variables)
target_word (i.e., the target word)
λ (i.e., the attenuation rate for file paths)

Output: variables with likelihood values

```
1  Function: get_similarity(groups, target_word, λ):
2  max_avg ← 0
3  foreach word_array : groups do
4      simi_group ← 0
5      foreach word : word_array do
6          d ← similarity(word, target_word)
7          increase simi_group by d ∗ λ
8      end
9      avg ← average(simi_group)
10     if avg > max_avg then
11     end
12     max_avg = avg
13 end
14 return max_avg
15 Function: NLP_main(variables, target_word):
16 foreach var : variables do
       /* for variable name.                                          */
17     var_name ← get_var_name(var)
18     simi_var ← get_similarity(var_name,target_word, 1)
       /* for type name.                                              */
19     type_name ← get_type_name(var)
20     simi_type ← get_similarity(type_name, target_word, 1)
       /* for function name.                                          */
21     func_name ← get_func_name(var)
22     simi_func ← get_similarity(func_name, target_word, 1)
       /* for file path.                                              */
23     path ← get_path(var)
24     simi_path ← get_similarity(path, target_word, 0.8)
25     var.likelihood ← (simi_var+simi_func+simi_type+simi_path)
26 end
```

4.5 Evaluation

Our evaluation seeks answers to the following questions:

Q1. Correctness: (a) Does VarSem infer variable usage semantics correctly? (b) How does combining the textual and context information of variables in VarSem programs affect the resulting correctness? (c) How does the correctness of VarSem compare to that of existing approaches?

Q2. Effectiveness: (a) How effective VarSem would be in helping software and security engineers to strengthen system security? (b) Are impact levels effective in controlling the correctness of inferring variable usage semantics?

Q3. Programming Effort: (a) How much programming effort does it take to express and execute VarSem programs? (b) How does this effort compare to using existing approaches?

4.5.1 Environmental Setup

The syntax of VarSem is defined by using the Scala embedded DSL facilities (Scala 2.12 and JDK 8). VarSem's preprocessing routines use Clang 6.0's *libtooling tool*. VarSem's textual information analysis uses Python 2.7, while its data-flow analysis routines use LLVM custom passes (LLVM public release 4.0). All experiments are performed on a workstation, running Ubuntu 16.04, 2.70GHz 4-core Intel i7-7500 CPU, with 15.4 GB memory.

4.5.2 Built-in Analysis Routines

VarSem comes with built-in analysis routines: an NLP-based textual information analysis and a data-flow analysis, whose implementations we discuss in turn next.

a) NLP-based textual information analysis. To compute the similarity between two words, we use Google's official pre-trained NLP model (Google News corpus with 3 billion running words [75]), and *Skip-Gram* algorithm implemented using *Word2vec*, a Google's word embedding tool [74].

b) Information-flow Analysis. An information flow analysis infers how a variable is used in terms of its flows from/to a specific function, its storing/reading into/from stable storage, and its comparison with other variables or constants. These inferencing tasks rely on traditional static analysis techniques, implemented as LLVM passes and Clang's analysis tools [45].

VarSem integrates these routines (i.e., a and b) in the same way as it would any new custom routines, thanks to its extensible architecture.

Table 4.1: Project Information

Project	Author Info	PII	LoC	Var. Usage	VarSem Rules
(1) MimiPenguin [79]	8 contributors	login user info	574	user	`$var.Name ~ "user"` `$var.Type ~ STRING`

(2) emailaddr for PostgreSQL [54]	1 contributor	email address	246	email address	`$var.Name ~ "email address"`
					`$var.Value <=>`
					`$var.Type ~ "emailaddr"`

(3) findmacs [51]	1 contributor	mac/ip address	515	mac/ip address	`$var.Name ~ "mac ip"` `$var.Type ~ STRING`
(4) emv-tools [30]	1 contributor	bank & credit info	9684 (1534)	smart card	`$var.Name ~ "smart card"` `$var.Value <-- "scard_connect"`
(5) ssniper [149]	Technology Services Group (UIUC)	SSN	2421	ssn	`$var.Name ~ "ssn"` `$var.Value <-- "represent_ssn"`
(6) phone number scanner [94]	1 contributor	phone number	381	phone number	`$var.Name ~ "phone number"` `$var.Value <-- "findPhoneNumber"`

(7) wdpassport-utils [115]	7 contributors	password	1504	password	`$var.Name ~ "password"`
					`$var.Value <-- "fgets_noecho"`
					`$var.Value <=>`
					`$var.Type ~ STRING`

(8) Biometric Authentication [100]	KYLIN Information Technology Co., Ltd.	biometric info	18305(1679)	bio feature	`$var.Name ~ "biometric feature"`
					`$var.Value <-- "bio_ops_get_feature_list"`
					`$var.Type ~ "feature_info"`

Notice: this table corrects the information presented in the corresponding table that appeared in [110], in which it misses the data types in cases (2) (7) (8) in column "VarSem Rules."

4.5.3 Evaluation Requirements

As a general solution for inferring variable usage semantics, it would be unrealistic to evaluate all possible scenarios in various domains. Hence, we constrained our evaluation to meet the

following requirements: (1) **Domains:** The evaluation scenarios should be representative of current real-world domains that need variable usage semantics analysis. (2) **Beneficiaries:** The evaluation scenarios should be helpful for software and security engineers in strengthening the protection of sensitive data. (3) **Scope:** The evaluation subjects should be real systems, with realistic variable usage cases. To meet these requirements, our evaluation focuses on the domain of personally identifiable information (PII), all data that can be used to identify a specific individual [125]. By compromising PII, attackers can threaten people's security and privacy. If VarSem can assist in identifying PII variables, developers would be able to adequately protect these variables' security and privacy.

a) to evaluate correctness, we use 8 open-source projects that manipulate PII, including user information, email addresses, MAC/IP addresses, bank/credit information, social security numbers, phone numbers, passwords, and bio-metric information ("PII" column in Table 6.2). *b) to evaluate effectiveness*, we use Yubico PAM [174], an open-source system for authenticating users that was subjected to two real security attacks (details in 4.5.4).

4.5.4 Evaluation Design

Correctness: In fact, PII variables differ greatly in their respective usage semantics, so each PII variable type requires a different VarSem program to identify it. As shown in Table 6.2, we wrote project-specific VarSem programs that infer given variable usage semantics (the "Var. Usage" column) by following their rules (the "VarSem Rules" column). Recall the syntax of VarSem rules: `$var.Name ~ "a"` means the variable's name matches "a", `$var.Type ~ "b"` means the variable's type matches "b", `$var.Value <=>` means the variable's value is compared, and `$var.Value <~~ "foo"` means the variable's value must have flown from function "foo". As an example, consider project "(8) Biometric Authentication" in which

VarSem infers those variables that store biometric features. The specified semantic category has two rules: (1) the variable's name matches "biometric feature" (textual information) and (2) its value should have flown from function `bio_ops_get_feature_list` (context information).

To evaluate correctness, we calculated the metrics of accuracy, precision, and recall. To obtain the ground truth, we recruited a volunteer (6+ years C/C++ experience) to manually find all variables that store the corresponding PII variables. To reduce the manual effort, we evaluated only the core modules of the larger projects 4 and 8 (their LoC metrics are parenthesized in the "LoC" column).

To answer question **Q1-a**, we executed each VarSem program to obtain the aforementioned metrics. To answer question **Q1-b**, we evaluated the cases in which semantic categories have only textual or context information. To answer question **Q1-c**, we evaluated the cases that use the simple string comparison function (i.e., `strcmp`[1]) to find target variables instead of VarSem's textual information analysis.

Without loss of generality, we assigned each rule the default impact value, while setting their likelihood thresholds to 0.6 and 0.8. That is, VarSem is to calculate the likelihood under the same impact level of each rule, and report only those variables whose likelihood of matching the rules is above 60% and 80% (we evaluated these two cases of likelihood independently).

Effectiveness: Our evaluation subject — Yubico PAM (four C source files, totalling 2486 lines of code) — has been subjected to two security attacks: one circumvented the authentication process through a particular password string [48], while the other leaked and tampered data through an unclosed file descriptor of the debug file [49]. Based on these two real-world attacks, we evaluate VarSem on the Yubico PAM codebase under two scenarios: (1) infer variables that store passwords, and (2) infer variables that store file descriptors of a debug file that has not been closed. By locating the variables in these scenarios, VarSem can help

[1] it returns the variables whose name matches the search pattern exactly.

software and security engineers to defend against the attacks above.

We wrote VarSem programs for the "password" and "debug file descriptor" scenarios. The first program's semantic category has 2 rules: (1) the variable's name matches "password" (i.e., textual information) and (2) it should be compared (i.e., context information). The second program's semantic category has 3 rules: (1) the variable's name matches "debug file" (textual information), (2) its type matches "FILE", and (3) it is not closed[2] (context information).

Similar to our evaluation of correctness, we also used accuracy, precision, and recall for evaluating effectiveness. Also, we used a volunteer to manually find all variables that store passwords and unclosed debug file descriptors to establish the ground truth.

To answer question **Q2-a**, we evaluated those VarSem programs whose semantic categories include both textual and context information. To answer question **Q2-b**, we evaluated the cases whose rules have different impact levels.

Programming Effort: To answer question **Q3-a**, we cou- nted the uncommented lines of code (ULoC) of VarSem programs for each scenario of our evaluation subjects. To answer question **Q3-b**, we counted ULoC for the same scenarios as the control group: (1) manually execute the required NLP and data-flow analysis routines; (2) package these routines as libraries, and use the Unix shell to invoke them.

4.5.5 Results

Correctness: The accuracy, precision, and recall results appear in Table 4.2. VarSem programs start their execution with the following data collection procedure, which entails

[2]With this attack fixed and all file descriptors correctly closed, we modified the file descriptors of debug file to remain open for our evaluation.

(1) extracting all program variables, producing the initial dataset, whose size is reported in the "Init." column; (2) pre-processing the initial dataset to eliminate duplicates, producing the pre-processed dataset, whose size is reported in the "Prepro." column.

Then, VarSem's execution differs based on the provided rules. For each of the 8 evaluation subjects, we provided a VarSem program containing both the textual and context information rules. For each program, we experimented with the likelihood thresholds of 0.6 and 0.8 ("THLD" column). The results reported in the "Textual+Context" column performed satisfactorily in *accuracy* (13 out of 16 times >= 80%), *precision* (13 out of 16 times > 80%), and passably in *recall* (8 out of 16 times > 60%). That is, VarSem infers variable usage semantics mostly correctly, with some PII variables missed (i.e., false negatives). These high false negatives are positively correlated with the specified likelihood thresholds: the larger the threshold, the less likely would VarSem designate variables as PII, with more *true* PII variables missed and smaller recall. Also, the larger the threshold, the higher the precision value. That is, when the threshold is large (0.8 in our case), VarSem indeed misses some *true* PII variables, but it correctly excludes many non-PII variables, so the false positives rate decreases, with the precision increasing.

Table 4.2: Results of Correctness

Proj.	Init.	Prepro.	THLD	(a) Textual Info only			(b) Context Info only			(c) Textual + Context			strcmp		
				accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall
(1)	115	89	0.6	95.5%	70.0%	87.5%	75.3%	26.7%	100%	**80.0%**	**30.1%**	**100%**	91.0%	NA	0%
			0.8	98.9%	100%	87.5%	75.3%	26.7%	100%	**98.9%**	**100%**	**87.5%**			
(2)	31	31	0.6	25.8%	0%	0%	83.9%	100%	73.7%	**74.2%**	**82.4%**	**73.7%**	38.7%	NA	0%
			0.8	29.0%	0%	0%	83.9%	100%	73.7%	**45.1%**	**100%**	**10.5%**			
(3)	97	75	0.6	80.0%	23.1%	37.5%	77.3%	32.0%	100%	**82.7%**	**38.1%**	**100%**	93.3%	100%	37.5%
			0.8	86.7%	37.5%	37.5%	77.3%	32.0%	100%	**93.3%**	**100%**	**37.5%**			
(4)	142	140	0.6	57.9%	7.9%	11.1%	95.0%	100%	74.1%	**95.0%**	**100%**	**74.1%**	80.7%	NA	0%
			0.8	79.3%	37.5%	11.1%	95.0%	100%	74.1%	**82.9%**	**100%**	**11.1%**			
(5)	620	323	0.6	96.6%	33.3%	10%	100%	100%	100%	**100%**	**100%**	**100%**	97.2%	100%	10%
			0.8	97.2%	100%	10%	100%	100%	100%	**97.2%**	**100%**	**10%**			
(6)	23	19	0.6	52.6%	45.5%	62.5%	84.2%	100%	62.5%	**84.2%**	**100%**	**62.5%**	57.8%	NA	0%
			0.8	52.6%	44.4%	50%	84.2%	100%	62.5%	**78.9%**	**100%**	**50%**			
(7)	205	190	0.6	95.8%	66.7%	22.2%	79.5%	17.4%	88.9%	**97.9%**	**85.7%**	**66.7%**	96.3%	100%	22.2%
			0.8	96.3%	100%	22.2%	96.7%	80.0%	44.4%	**95.8%**	**100%**	**11.1%**			
(8)	579	569	0.6	87.3%	0%	0%	99.6%	100%	50%	**99.3%**	**50%**	**50%**	99.3%	NA	0%
			0.8	92.4%	0%	0%	99.6%	100%	50%	**99.6%**	**100%**	**50%**			

Also, we modified its VarSem program to produce 2 variants: keep only the textual informa-

tion rules (case-a, the "Textual Info only" column) and keep only the context information rules (case-b, the "Context Info only" column). Then we compared VarSem's output for these two variants with that of the original program, which includes both the textual and context information rules (case-c, the "Textual + Context" column). Case-c outperformed (or performed equal to) the other two cases (14 out of 16 times better than or equal to case-a while 9 out of 16 times better than or equal to case-b. 5 out of 8 times better than or equal to both case-a and b for the threshold 0.6). That is, considering both the textual and context information can increase VarSem's performance. However, if the performance of case-a or b is poor, case-c's performance deteriorates as well. For example, in the project-2 "emailaddr for PostgreSQL", because its case-a generates unsatisfactory results (accuracy $< 30\%$, precision and recall $= 0\%$), case-c performs no better than case-b. Recall that we assigned the same impact levels to all rules. By adjusting the rules' impact levels to fit specific variable usage semantics, the performance of case-c can be improved (we evaluate how impact levels work below).

We also evaluated the performance of a simple string comparison function (the "strcmp" column) to process textual information rules. Overall, its performance is worse than any of the cases a, b, and c above. In five out of eight projects (i.e., projects 1, 2, 4, 6, and 8), `strcmp` 's inference results are useless (i.e., the precision and recall values are either NA—the number of true/false positives is "0" or 0%—the number of true positives is "0"). This result reveals that the symbolic names of the PII variables in these projects are not descriptive of their usage semantics. For example, in the project-4 "phone number scanner", the variables that store phone numbers are named as "inStr" and "outputStr" rather than anything resembling "phone_number." However, because of VarSem's NLP-based analysis for rules of textual information, VarSem's results (i.e., "Textual Info only" column) for these projects always outperform (or perform equal to) the strategy of comparing strings for equality.

Table 4.3: Results of Effectiveness

Var. Usage	Rules	Impact Lv.	THLD	Textual + Context		
				accuracy	precision	recall
password	(a)`$var.Name~"password"` (b)`$var.Value<=>`	a<b (a=5, b=10)	0.6	40.60%	4.20%	100%
			0.8	86.90%	15.80%	92.30%
		a=b (a,b=10)	0.6	60.60%	5.80%	92.30%
			0.8	95.60%	32%	61.50%
		a>b (a=10, b=5)	0.6	86.90%	15.80%	92.30%
			0.8	97.60%	53.80%	53.80%
debug file	(a)`$var.Name~"debug file"` (b)`$var.Type-FILE` (c)`$var.Value!~>"fclose"`	a<b=c (a=5, b,c=10)	0.6	98.20%	100%	18.20%
			0.8	97.80%	NA	0%
		a=b=c (a,b,c=10)	0.6	100%	100%	100%
			0.8	97.80%	NA	0%
		a>b=c (a=10, b,c=5)	0.6	100%	100%	100%
			0.8	97.80%	NA	0%

Effectiveness: Table 4.3 shows the effectiveness results of inferring the variable usage semantics scenarios, described as "password" and "debug file" (the "Var. Usage" column). Overall, VarSem performed satisfactorily in inferring the variables that store unclosed debug file descriptors (100% accuracy, precision, and recall in the best case), and passably in inferring the variables that store passwords (97.6% accuracy, 53.8% precision, and 53.8% recall in the best case). Hence, based on the returned variable lists, developers can directly locate the unclosed debug file descriptors and reduce the manual effort required to locate password variables. VarSem performed less effectively in the "password" scenario, as it was given more general rules. Indeed, in the "Yubico Pam" project, in addition to password variables, other variables are semantically related to "password" and are compared for equality. In contrast, only those variables that store unclosed debug file descriptors have a symbolic name related to "debug file", the type matching "FILE", and the value never flowing to function `fclose`.

When evaluating the influence of impact levels in the "Yubico Pam" subject, increasing the impact level of textual information rules does improve VarSem's performance, as the given textual information rules are more descriptive of the target semantics than the context information rules. That is, many variables compared for equality may not store passwords (the "password" case), and many variables never flowing to `fclose` function may not store

debug file descriptors. Since the likelihood values are weighted by their impact levels, the textual information with impact levels higher than those of the context information would contribute more to the final results, resulting in better accuracy/precision/recall.

Programming Effort: Table 4.4 shows the evaluated effort as hand-written ULoC. For all evaluated subjects, it took only ≈6 ULoC to write a project-specific VarSem program. However, to accomplish the same inferencing task without VarSem took ≈2,500 ULoC of analysis routines written in C and Python. Even if these routines were packaged as libraries, invoking them would take ≈16 ULoC of Unix shell scripts. Without VarSem, developers would need to possess specialized expertise: (1) a familiarity with all relevant libraries/routines and their interactions, and (2) an understanding of the output format of each routine involved. In contrast, with VarSem, developers specify a semantic category using high-level declarative rules, thus lowering the programming effort.

Table 4.4: Programming Efforts of VarSem

Languages	Inference Tasks
VarSem	∼6 (ULoC)
Unix shell	∼16 (ULoC)
All relevant routines	2442 (ULoC)

4.5.6 Discussion

Rules. As the effectiveness results demonstrate (4.5.5), specific rules outperform general rules. Although easier to express, general rules (e.g., a variable type is a string) match too many irrelevant variables, which more specific rules can effectively filter out.

Thresholds. As the correctness results demonstrate (4.5.5), precision and recall can be improved by modifying thresholds: in general, a high threshold increases precision, while a low threshold increases recall. This behavior is due to higher thresholds causing fewer

reported variables, thus decreasing false positives. In contrast, lower thresholds cause more reported variables, thus decreasing false negatives. With the thresholds being in the range of [0,1], choosing the thresholds of 0.6 and 0.8 reports the variables whose likelihoods of matching the rules are above 60% and 80%, respectively.

Impact levels. It would be unrealistic to expect programmers to follow the prescribed naming convention all the time. Hence, a good practice for VarSem users is to pre-check the presence of and adherence to a naming convention. Depending on the findings, the VarSem performance can be improved by adjusting rules' impact levels: in the presence of unsystematic naming practices, reduce the impact levels of textual information rules, and vice versa.

Textual Information Weights. As discussed in the sections above, variable usage semantics analysis involves four kinds of textual info: variable name, type name, function name, and file path. We conjecture that each of these textual pieces of info contributes the variable usage semantics dissimilarly. To verify this conjecture, we carried out another evaluation: calculate the variable's likelihood value based on each of these pieces in isolation (i.e., columns "(a) Var Name only", "(b) Function Name only", "(c) Type Name only", and "(d) File Path only"). Then we compared the accuracy, precision, and recall of these four cases.

The results in Table 4.5 show two insights: (1) The cases (a), in general, outperform the rest of the cases, thus indicating that variable name contribute more to the variable usage semantics than function name, type name, and file path. At the same time, the results from file path contribute little value (i.e., the most of "NA" values). One explanation for this finding it that variable names can uniquely identify variables, while the rest of variable properties only provide additional information that is not unique. (2) However, which kind of textual info is more valuable depends on the target program's implementation practices.

For example, if the program's function names, composited type names, or file names are meaningfully named, while its variable names are not so, then the former should be considered more than the latter. Such a case can arise when a project is developed starting from a high-level skeleton that comprises only types and function names, so individual developers can then fill in the skeletal representation with concrete code. Hence, to determine how to leverage each of these textual information pieces optimally, developers should not only consider the general variable naming trends, but also the specific naming practices of the analyzed projects. We will revisit these insights when describing TEE-DRUP in Chapter 6.

Table 4.5: Textual Information Weights

Proj.	THLD	(a) Var Name only			(b) Function Name only			(c) Type Name only			(d) File Path only		
		accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall
(1)	0.6	98.9%	100%	87.5%	69.7%	22.9%	100%	98.9%	100%	87.5%	69.7%	22.9%	100%
	0.8	98.9%	100%	87.5%	80.9%	15.4%	25%	98.9%	100%	87.5%	69.7%	22.9%	100%
(2)	0.6	29.0%	33.3%	15.8%	35.5%	44.4%	21.1%	38.7%	NA	0	38.7%	NA	0
	0.8	32.3%	25%	5.3%	32.3%	0	0	38.7%	NA	0	38.7%	NA	0
(3)	0.6	86.7%	37.5%	37.5%	50.7%	10.9%	50%	77.3%	0	0	89.3%	NA	0
	0.8	93.3%	100%	37.5%	56%	12.1%	50%	80%	0	0	89.3%	NA	0
(4)	0.6	75.7%	23.1%	11.1%	50.7%	2.3%	3.7%	78.6%	0	0	42.1%	20.7%	70.4%
	0.8	82.9%	100%	11.1%	52.1%	2.4%	3.7%	80%	0	0	78.6%	38.5%	18.5%
(5)	0.6	97.2%	100%	10%	96.3%	45.5%	100%	88.2%	0	0	91.6%	27%	100%
	0.8	97.2%	100%	10%	95.0%	33.3%	60%	88.2%	0	0	91.6%	27%	100%
(6)	0.6	52.6%	0	0	47.4%	40%	50%	89.5%	80%	100%	36.8%	37.5%	75%
	0.8	52.6%	0	0	47.4%	40%	50%	68.4%	100%	25%	36.8%	37.5%	75%
(7)	0.6	96.3%	100%	22.2%	88.4%	19.0%	44.4%	73.7%	0	0	95.3%	NA	0
	0.8	96.3%	100%	22.2%	91.1%	16.7%	22.2%	76.8%	0	0	95.3%	NA	0
(8)	0.6	97.4%	0	0	51.8%	0	0	87.7%	5.4%	100%	76.8%	0	0
	0.8	98.9%	0	0	62.4%	0	0	88.0%	0	0	76.8%	0	0

Runtime. VarSem programs execute in time proportional to the size of target codebases and the complexity of rules: it takes longer to analyze larger projects with complex rules. In our evaluation, the VarSem executes within an acceptable time: less than 1 minute (projects 1,2,3,6), less than 3 minutes (project 4), and less than 6 minutes (projects 5,7,8). The most time-consuming tasks are the NLP-based textual information analysis and data-flow analysis.

Applicability. Our built-in data-flow analysis routines are LLVM-based, thus limiting their applicability to C/C++ projects. Hence, our evaluation subjects are written in C/C++. However, VarSem's applicability can be extended to other languages thanks to its con-

tainerized deployment model (4.3.2). When it comes to VarSem's syntax, it would need to change to accommodate how the target language expresses variables. For example, in dynamically typed languages, variable types are determined and can change at runtime, rendering VarSem's static type-related rules (e.g., `$var.Type ~ "int"`) inapplicable. Nevertheless, due to VarSem's extensible architecture and meta-programming support, developers can introduce rules with suitable operator/operands and analysis routines that can handle new analysis scenarios (e.g., in a dynamically typed language, the type can be bound to numeric values: `$var.DynType ~ "numeric"`)

Threats to validity & Limitations. The internal validity is threatened by comparing VarSem's results with those of `strcmp` rather than of regular expressions. Two observations mitigate this threat: (1) `strcmp` provides a standard comparison baseline; (2) as discussed in chapter 3, we assume user familiarity only with variable usage semantics (VUS), not expecting them to be able to list all variations of symbolic names. For example, a user would be looking for the "password" VUS, being unaware which symbolic names ("pwd", "p_wd", "p_w_d", etc.) to enumerate with a regular expression.

The external validity is threatened by evaluating only with eight third-party C/C++ subjects (see 4.5.3). Although covering various PII variable scenarios, these projects cannot represent all possible scenarios. Further, our findings might not generalize to other languages, and only additional studies can mitigate this threat. We plan to open-source VarSem and our experiments, so others could conduct studies with different subjects and settings.

For limitations, although VarSem outperforms `strcmp` in most cases, `strcmp` can search projects with highly descriptive variable names with acceptable accuracy/precision and passable recall (i.e., rows (3)(5)(7) and "strcmp" column in Table 4.2). As VarSem typically executes in seconds or minutes at worst, `strcmp` can execute in a second or less, achieving comparable accuracy/precision/recall. In addition, extending VarSem to support new operators/operands can

require changing the runtime, incurring an additional programming effort, amortized only by subsequent uses of the extension. However, with sufficient domain knowledge, VarSem users should be able to select the most suitable analysis algorithms and meaningfully extend VarSem.

4.6 Conclusion

This chapter has presented *variable usage semantics analysis*, reified as a DSL—VarSem—with a novel NLP-based analysis. VarSem features declarative and customizable rules bound to analysis routines for inferring both textual and context variable information. By reducing the effort of variable usage semantics analysis, VarSem can benefit both developers and security analysts.

Chapter 5

Isolating Sensitive Data and Functions

The execution of mission-critical real-time systems must comply with real-time constraints. Many such systems also contain vulnerable sensitive program information (SPI) (i.e., sensitive algorithms and data) that must be protected. Failing to satisfy either of these requirements can lead to catastrophic consequences. Consider using an autonomous delivery drone to transport packages, containing food, water, medicine, or vaccines, to remote and hard-to-reach locations. Emergency personnel and professional nature explorers often depend on drone delivery when dealing with various crises. The drone's navigation component has real-time constraints; if it fails to compute the instructions for the autopilot to adjust the flight's directions or airspeed in a timely fashion, the drone may become unable to adjust its trajectory properly and deviate from the programmed delivery route. Since the cargo often must be delivered under strict time requirements, deviating from the shortest route can cause the entire delivery mission to fail. In addition, the software controlling module (e.g., navigation) constitutes sensitive program information (SPI). If an ill-intentioned entity takes control over the module's execution, the entire drone can be misrouted, causing the delivery to fail. Irrespective of the causes, the consequences of a failed delivery can be potentially life-threatening.

The vulnerabilities above can be mitigated by isolating SPI functions in a secure execution environment that would also control their interactions with the outside world. As a way to realize this idea, hardware manufacturers have started providing trusted execution en-

vironments (TEEs), special-purpose processors that can be used to execute SPI-dependent functionality. TEE can reliably isolate trusted code (i.e., in the secure world) from regular code (i.e., in the normal world); the secure world comes with its own trusted hardware, storage, and operating system. A special communication API is the only avenue for interacting with TEE-based code. With the TEEs being hard to compromise, isolating SPI in the secure world effectively counteracts adversarial attacks and prevents intellectual property theft. However, to benefit from trusted execution, systems must be designed and implemented to use different implementations of the TEE (e.g., OP-TEE [128], SGX [47]). Adapting existing real-time systems to use the TEE requires non-trivial, error-prone program transformations, while the transformed system's execution must continue to adhere to the original real-time constraints.

In particular, a developer transforming a system to take advantage of the newly introduced TEE module requires undertaking the following tasks: 1) isolate SPI-dependent code; 2) redirect invocations of SPI functions to TEE communication API calls; 3) verify that the transformed system continues to meet the original real-time constraints. Notice that all of these tasks are hard to perform correctly by hand.

To complete task 1), a developer not only needs to correctly extract the SPI-dependent code from the system, but also correctly identify all the dependencies; due to the potential complexity of these dependencies, some SPI-dependent code cannot be isolated in TEEs. Most importantly, different TEEs (e.g., OP-TEE and SGX) expose dissimilar APIs and conventions for isolating SPI functions. A SPI-dependent function can be isolated in both TEE implementations, only one of them, or neither of them. To determine how a SPI function can be isolated, developers must be intimately familiar with both the original source code and the requirements of each TEE implementation. As is often the case, developers performing adaptive maintenance are often not the ones who wrote the original system. To facilitate this

difficult and error-prone process, prior work has proposed automatic program partitioning, even in the presence of pointer-based function parameters [106]. However, this prior work leaves out the issues of verifying whether a given partitioning strategy is valid or whether the partitioned system would comply with the real-time constraints.

To complete task 2), the developer must write by hand the communication logic required for the normal and secure worlds to talk to each other, correctly applying suitable TEE APIs that establish customized communication channels. However, to accomplish this task correctly, developers must invest a great deal of time and effort to learn and master both the OP-TEE or SGX implementations: the OP-TEE provides more than 130 APIs and about 40 data types [66, 67, 68], while SGX provides an Enclave Definition Language (EDL) with more than ten syntactic categories [90].

To complete task 3), the developer must be willing to develop additional test cases that can verify whether the transformed system satisfies the original real-time constraints. Existing approaches take advantage of profiling tools, including Pin tool [113] and gperftools [77], which require that profiling probes be added by hand.

To facilitate the process of adapting real-time systems to protect their SPI-dependent code using a TEE, this article presents RT-Trust, a program analysis and transformation toolset that supports developers in partitioning C-language systems in the presence of real-time constraints. The developer can either specify the TEE implementation (i.e., OP-TEE or SGX) as a compiler option, or rely on RT-Trust to automatically determine the available implementation by inspecting the system. Through a meta-programming model, the developer annotates individual C functions to be isolated into the secure world. Based on the annotations, the RT-Trust static and dynamic analyses determine whether the suggested partitioning strategy is feasible, and whether the partitioned system would comply with the original real-time constraints for both the OP-TEE or SGX. A continuous feedback

loop guides the developer in restructuring the system, so it can be successfully partitioned. Finally, RT-Trust transforms the system into the regular and trusted parts, with custom generated TEE-specific communication channel between them. If the transformed code fails to meet real-time constraints, it raises custom-handled exceptions. RT-Trust reduces the programmer effort required to partition real-time systems to take advantage of the emerging TEEs.

The contribution of this work is four-fold:

1. **A Fully Declarative Meta-Programming Model** for partitioning real-time systems written in C to take advantage of the TEEs; the model is realized as domain-specific annotations that capture the requirements of different partitioning scenarios.

2. **Static and Dynamic Checking Mechanisms** that identify whether a system can be partitioned as specified for a given TEE implementation, and how likely the partitioned version is to meet the original real-time constraints. The analyses integrate a feedback mechanism that informs developers how they can restructure their systems, so they can be successfully partitioned.

3. **RT-Trust Refactoring**, a compiler-based program transformation for C programs that operates at the IR level, while also generating customized communication channels and real-time deadline violation handling.

4. **A Platform-Independent Metric** for assessing by how much a SPI function is expected to degrade its performance once moved to the TEE, and comparing such degradations between different TEEs; we evaluate the applicability of this metric on five classic security algorithms and two critical functions in a popular drone controller system.

To concretely realize our approach, we have created RT-Trust as custom LLVM passes and runtime support. Our evaluation shows that RT-Trust saves considerable programmer effort by providing accurate program analyses and automated refactoring. RT-Trust's profiling facilities also accurately predict whether refactored subjects would continue meeting real-time constraints.

5.1 Solution Overview

In this section, we introduce the toolchain of our compiler-based analyzer and code refactoring tool, and then we describe the input and output of RT-Trust.

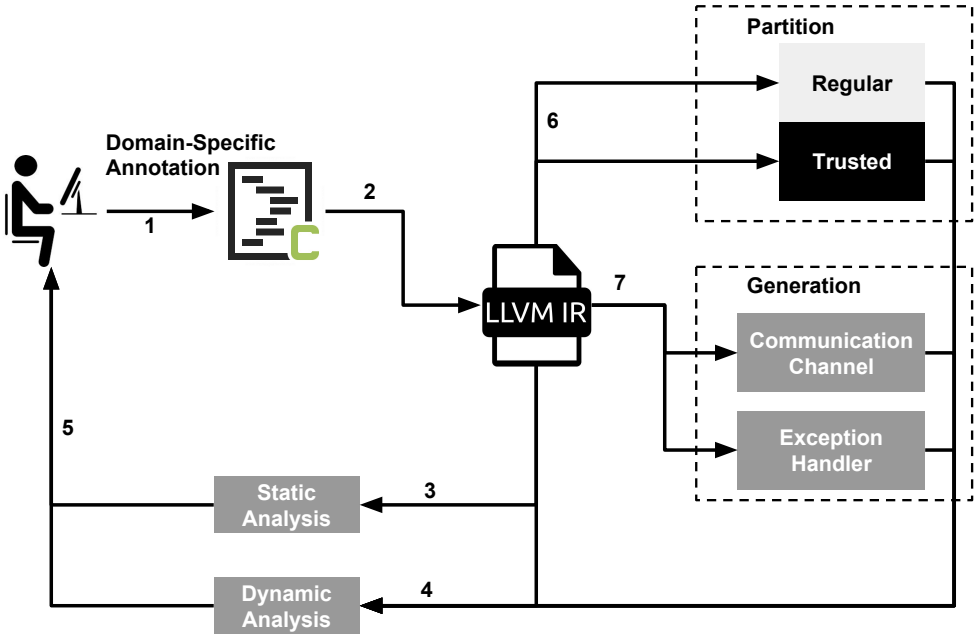

Figure 5.1: The RT-Trust Process

5.1.1 Software Development Process

Figure 5.1 describes the software development process of using RT-Trust to partition real-time systems to take advantage of TEEs. Given a real-time system, the developer first specifies the SPI-dependent functions in the source code using the RT-Trust domain-specific annotations (DSA) (step 1). The annotated source code is then compiled to LLVM intermediate representation (IR). The compilation customizes Clang to specially process the DSA metadata (step 2). After that, RT-Trust determines whether the TEE is implemented as OP-TEE or SGX by inspecting the execution environment or the build configuration. To check whether the specified partitioning scenario can be realized, RT-Trust statically analyzes the system's call graph (step 3). Given the system's call graph and a partitioning specification, RT-Trust constructs the partitionable function graph (PFG), which contains all the information required to determine if the specification is valid. While static analysis determines the semantic validity of a partitioning specification, a separate dynamic analysis phase estimates whether the partitioned system would continue complying with the original real-time constraints. To that end, RT-Trust instruments the system by inserting probes at the IR level (step 4). The inserted probes estimate the partitioning scenarios' memory consumption and function invocation latencies. The system is then exercised under expected loads. The results are then reported back to the developer (step 5). This prior analysis and validation routines make it possible for the developer to modify the original system make it possible to move the SPI functions to execute in the secure world. Once the developer determines that the system can be partitioned with satisfying performance, RT-Trust then automatically divides the system's IR into regular and trusted parts (step 6). The former will be run in the normal world, while the latter in the secure world. To enable these two portions to communicate with each other, RT-Trust generates communication channels customized for OP-TEE and SGX. In addition, to handle the violations of real-time constraints,

RT-Trust generates exception handling code (step 7). Notice that all these code generation processes are configured entirely by the DSAs applied to the system's SPI functions. Having undergone a partitioning, the system then goes through the final round of verification by dynamically profiling the partitioned system (step 4). The profiling identifies the performance bottleneck while estimating whether the transformed system continues to satisfy the real-time constraints (step 5). Finally, RT-Trust generates a descriptive report that includes the outcomes of various profiling scenarios and suggestions for the developer about how to remove various performance bottlenecks.

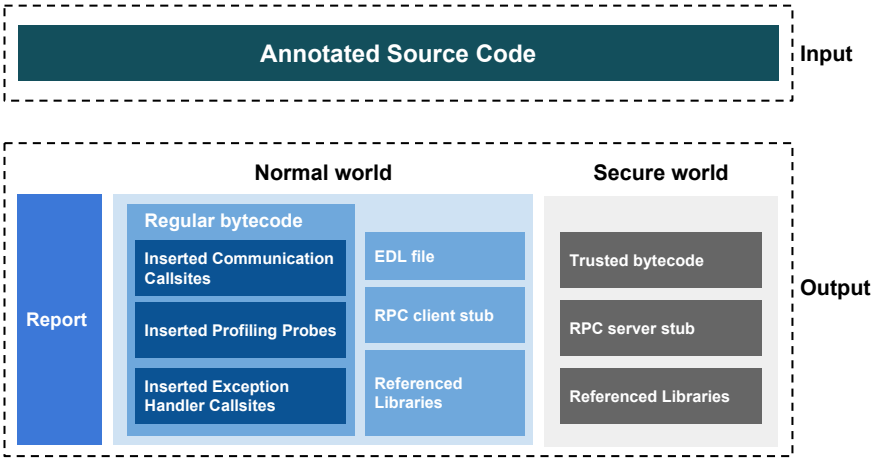

Figure 5.2: RT-Trust's Input and Output

5.1.2 Code Transformation and Generation

Figure 5.2 shows RT-Trust's code transformation and generation. As input, RT-Trust receives the annotated source code. As output, it transforms the IR of the input source and also generates additional code that is compiled and integrated into the normal and secure world partitions. For the normal world, RT-Trust transforms the IR by inserting profiling probes, exception handlers, and communication channels. All generated code can be further customized by hand if necessary. The transformed IR code, generated source code (i.e., RPC

client stub for OP-TEE and an EDL file for SGX), and referenced libraries (e.g., encryption, profiling) are eventually linked with the normal world's executable. Similarly, for the secure world, the trusted IR, RPC server stub (for OP-TEE), and the referenced libraries are linked with the secure world's executable, which can run only in the secure world of TEE.

5.2 Meta-programming Model

To accommodate application programmers, RT-Trust follows a declarative programming paradigm, supported by a meta-programming model. This model makes use of the annotation facility recently introduced into the C language. A C programmer can annotate functions, variables, parameters, and code blocks to assign a customized semantics. The semantics is realized by the compiler by means of a special processing plug-in. For example, if a function is annotated with `nothrow`, the compiler can check that the function contains no statement that can raise exceptions; if the check fails, an informative message can be displayed to the programmer, who then can modify the function's code accordingly. Despite the large set of built-in Clang annotations [150], none of them are designed for real-time systems and TEE.

For our meta-programming model, we design and implement a set of domain-specific annotations that describe the real-time constraints, code transformation and generation strategies required to automatically transform a real-time system, so its subset can be partitioned to TEE for trusted execution. We call our domain-specific annotations Real-Time Trust Annotations, or RTTA for short. We integrate RTTAs with the base Clang annotation system, so the compiler can analyze and transform real-time systems, as entirely based on the declarative annotations, thus reducing the development burden by enabling powerful compiler-based code analysis and transformation. In this section, we first describe the gen-

eral syntax of RTTAs. Then, we introduce each annotation and its dependencies in turn. Finally, we illustrate how to use these annotations through an example.

5.2.1 General Syntax

In the code snippet below, RTTA follows the GNU style [69], one of the general syntaxes supported by Clang. The form of attribute specifier is `__attribute__ ((annotation-list))`. The annotation list (`<annotation-list>`) is a sequence of annotations separated by commas. Each annotation contains the annotation name and a parenthesized argument list (`<annotation-list>`). An argument list is a possibly empty comma-separated sequence of arguments.

```
1 __attribute__((<annotation-list>))
2 <annotation-list> ::= <annotation>,<annotation>*
3 <annotation> ::= name (argument-list)
4 <argument-list> ::= <argument>,<argument>*
5 <argument> ::= various arguments
```

5.2.2 Code Partition Annotation

The code partition annotation informs RT-Trust to perform two tasks: 1) analyze the validity of partitioning for each annotated function, and 2) extract the annotated functions that can be partitioned from the source code. The annotation `partition` can be applied to any declared function in the source code, and takes no arguments, as follows:

```
1 __attribute__((partition))
```

5.2.3 Code Generation Annotations

Code generation annotations that appear in the code snippet below enable developers to customize 1) a specific communication mechanism (e.g., RPC) for the normal and secure worlds to talk to each other, and 2) an exception handler for handling the cases of violating real-time constraints when executing a partitioned system. When annotating with `rpc`, the developer can specify the `shared_memory` or `socket` options as the underlying RPC delivery mechanism. The data transferred between the partitions can be specified to be encrypted or compressed by using the `yes` and `no` options. Note that the `rpc` annotation applies only to OP-TEE to specify how to generate RPC stubs. For SGX, RT-Trust instead generates an EDL file and proxy functions. By annotating pointer and array parameters with `paramlen`, the developer can indicate their length. The `<length>` attributes are used by the marshaling and unmarshaling phases on the communication channels. For the pointer parameters, the `<length>` attribute reports the size of the data the pointer is referencing. Although recent advances in complex static analysis make it possible to automatically infer the size of pointer-based parameters [106], our design still relies on the programmer specifying the length information by hand. This design choice allows for greater flexibility. The `paramlen` annotation makes it possible for the developer to reserve the required amount of space for the annotated parameters, and then specify how to generate customized marshaling and unmarshaling code. If the developer also annotates that function with `memsize`, the RT-Trust dynamic analysis suggests an approximated length value (details appear in Section 5.3.2). By annotating with `exhandler`, the developer can specify how to handle the exceptions potentially raised by the annotated function. The annotation has three parameters: a handler function's name (`<method>`), the target's real-time constraints (`<constraint_type>`), and the trigger threshold (`<times>`) (i.e., the number of times an annotated function can violate the target constraints before triggering the handler function). We explain how RT-Trust generates code, as based

on these annotations, in Section 5.4.

```
__attribute__((rpc(<type>, <encryption>, <compression>)))
<type> ::= shared_memory | socket
<encryption> ::= yes | no
<compression> ::= yes | no

__attribute__((paramlen(<length>)))
<length> ::= n (n is integer, n > 0)

__attribute__((exhandler(<times>, <method>, <constraint_type>)))
<times> ::= n (n is integer, n > 0)
<method> ::= "default" | method name (string)
<constraint_type> ::= exetime | period | memsize
```

5.2.4 Profiling Annotations

The annotations in the code snippet below configure the RT-Trust profiler to determine if a partitioned system would still meet the original real-time constraints.

Profiling Real-Time Constraints: RTTA provides three annotations for profiling to determine whether given real-time constraints would remain satisfied: 1) `exetime` (i.e., execution time), 2) `period`, and 3) `memsize` (i.e., memory consumption). The `<type>` argument specifies whether the constraint is `hard` or `soft`. The `hard` mode means that violating the constraint is unacceptable, while the `soft` mode means such violations, to some extent, can be accepted. Based on these types, the profiler reports whether the annotated function can be transformed for trusted execution, without violating the specified real-time constraints. For the execution time attribute, the developer can specify the profiling method (i.e., `timestamping`

and `sampling`) and the completion deadline (i.e., `<deadline>` to meet. For `period`, one can specify the time interval between invocations of a SPI function. For memory consumption, the memory size can be limited by setting an upper-bound via the `<limit>` argument.

```
1  __attribute__((exetime(<type>, <method>, <deadline>)))
2  <type> ::= hard | soft
3  <method> ::= timestamping | sampling
4  <deadline> ::= n (n is integer, n > 0)
5
6  __attribute__((period(<type>, <interval>)))
7  <type> ::= hard | soft
8  <interval> ::= n (n is integer, n > 0)
9
10 __attribute__((memsize(<type>, <limit>)))
11 <type> ::= hard | soft
12 <limit> ::= n (n is integer, n > 0)
```

5.2.5 RTTA Dependencies

As compared to the annotations that can be specified independently (e.g., `partition`, `rpc`, and the profiling annotations), other annotations must be specified with their dependencies. For example, the annotation `paramlen` cannot be specified, unless `rpc` also appears among the function's annotations. The `paramlen` annotation is used for generating the marshaling and unmarshaling logic of the communication channels. Likewise, without annotations specifying real-time constraints, the exception handling code is unnecessary: `exhandler` must come together with real-time constraint annotations. The RT-Trust analysis process checks the adherence to these domain-specific semantics of RTTA and reports the detected violations.

5.2.6 RTTA in Action

Consider the example originally described in the beginning of this chapter: a drone navigates, with its autopilot continuously obtaining the current geolocation from the GPS sensor to adjust the flying trajectory in a timely fashion. The function of obtaining geolocations is SPI-dependent, and as such should be protected from potential interference by placing it in the secure world. To that end, the developer annotates that function, informing RT-Trust to transform the code, so the function is separated from the rest of the code, while also generating the necessary code for communicating and exception handling. Optionally, the system can be annotated to be profiled for the expected adherence to the original real-time constraints after it would be partitioned. The function `getGPSLocation` annotated with RTTAs appears below. Based on these annotations, our customized Clang recognizes that the function needs to be partitioned and moved to the secure world (`partition`). Meanwhile, RT-Trust will generate a communication channel over shared memory with the encrypted and compressed transferred data between the partitions (`rpc`). In addition, during the marshaling and unmarshaling procedure, the allocated memory space for the function's parameter will be 100 bytes (`paramlen`). Further, RT-Trust will insert the measurement code to profile the function's real-time constraints. It instruments the function's execution time with the `timestamping` algorithm and `hard` mode to check whether it meets the deadline (20 ms) (`exetime`), and checks whether the invocation interval would not exceed 50 ms (`period`). It estimates the memory consumption, and checks whether it exceeds 1024 bytes in the `soft` mode (`memsize`). Finally, if the real-time deadline constraint has been broken more than once, it will be handled by the exception handler function "myHandler" (`exhandler`). The declarative meta-programming model of RT-Trust automates some of the most burdensome tasks of real-time system profiling and refactoring. In the rest of the manuscript, we discuss some of the details of the RT-Trust profiling, code transformation, and code generation infrastructure.

```
1  Location loc; // global variable
2  Location getGPSLocation // SPI function
3  (GPSState * __attribute__((paramlen(100))) state)
4  __attribute__(( partition,
5      rpc(shared_memory, yes, yes),
6      exhandler(1, "myHandler", exetime),
7      exetime(hard, timestamping, 20),
8      period(hard, 50),
9      memsize(soft, 1024) )) {...}
10 // adjusting Drone direction
11 void adjustDirection(Location l) {...}
12 void fly() {
13     loc = getGPSLocation(state);
14     adjustDirection(loc);
15 }
16
17 int main() {
18 fly(); ... }
```

5.3 Analyses for Real-Time Compliance

The automated refactoring described here has several applicability limitations. One set of limitations stems from the structure of the system and its subset that needs to be moved to the trusted partition. Another set of limitations are due to the increase in latency that results in placing a system's subset to the trusted execution zone and replacing direct function calls with RPC calls. The increase in latency can cause the system to miss its real-time deadlines, rendering the entire system unusable for its intended operation. To check if the structure of

the system allows for the refactoring to be performed, RT-Trust features a domain-specific static analysis. To estimate if the refactored system would still meet real-time requirements, RT-Trust offers several profiling mechanisms, which are enabled and configured by means of RTTAs.

5.3.1 Static Analysis

The TEE implementation in place (i.e., OP-TEE or SGX) determines whether RT-Trust can realize a given partitioning scenario. That is, a scenario may work on the OP-TEE but not on the SGX, and vice versa. To that end, RT-Trust not only allows the developer to specify the TEE implementation, but it also automatically inspects the compilation environment to determine the TEE implementation. After that, RT-Trust checks whether the scenario adheres to the following three rules, referred to as `zigzag`, `pointers`, and `global variable`. If the code passes all three checks, RT-Trust can successfully carry out the specified partitioning scenario. A failed check report identifies why the code needs to be refactored to make it amenable to partitioning.

Zigzag Rule: Consider a set of functions T_1, annotated with the `partition` annotation, and another set of functions T_2, containing the rest of all the functions. The zigzag rule defines the restrictions imposed by different TEEs:

For OP-TEE, the zigzag rule states that functions in T_2 cannot invoke functions in T_1, as such invocations would form a zigzag pattern. This restriction is caused by the strict one-way invocation of the functions in the trusted zone from the normal world. The normal world can call functions in the trusted zone, but not vice versa. One can fix violations of the zigzag rule by annotating the offending function, called from the trusted zone, with `partition`, so it would be placed in the trusted partition as well, so it would be invocable via a local function

call. Our assumption of relying on the static version of the call graph is reasonable for the target domain of real-time systems written in C, in which functions are bound statically to ensure predictable system execution.

For SGX, the zigzag rule states that even though functions in T_2 can invoke functions in T_1, such invocations must be restricted to some small number (i.e., threshold) due to the high communication latency between the normal and secure worlds. That is, although SGX supports the zigzag calls, the program performance suffers from the high latency of such invocations [114]. One can tune the threshold to balance the trade-off between efficiency and utility. Once the threshold comes to "0", the zigzag rule regresses to the one used for OP-TEE.

Global Variable Rule: Since the partitioning is performed at the function level, the distributed global state cannot be maintained. As a result, each global variable can be placed either in the normal or trusted partition and accessed locally by its co-located functions. Violations of this rule can be easily detected. One exception to this rule is constant global variables, which due to being unmodifiable can be replicated across partitions.

Pointers Rule: The pointers rule restricts the types that can be used as parameters of the partitioned functions: 1) function pointers and pointer arrays cannot be passed as parameters, and 2) struct parameters cannot contain pointer members. For SGX, RT-Trust strictly enforces this rule, as the SGX Enclave Definition Language (EDL) has no support for such pointer types. However, for OP-TEE, only function pointers cannot be supported. For their code to abide by this rule, developers can refactor the target program, so the partitioned functions take no such pointer parameters. Alternatively, developers can manually implement specialized logic for marshaling/unmarshaling these parameters.

Partitionable Function Graph: To check the above rules, RT-Trust introduces a partitionable function graph (PFG). This data structure extends a call graph with special markings for the functions that can be partitioned. To construct a PFG, RT-Trust starts by walking the call graph for the functions annotated with `partition`. By checking whether these functions comply with the zigzag and global variable rules, it removes the function nodes that break these rules. The resulting graph is the PFG.

Specifically, RT-Trust sets each function annotated with `partition` as the root function, and then traverses its subgraph. During the traversal, RT-Trust checks whether all subgraph elements are also annotated with `partition`. If so, RT-Trust adds the entire subgraph to the PFG, and then moves to the next annotated function. After examining the zigzag rule, the PFG contains several sub-callgraphs of non-zigzag functions annotated to be partitioned. Next, RT-Trust collects global variable information for each function already in the PFG. It then examines whether the variables are operated by the functions in the PFG only. If so, RT-Trust adds these functions to the PFG. Otherwise, RT-Trust removes the entire subgraph containing the violating function from the PFG. The final PFG contains all the necessary information (e.g., global variables, parameters, and annotations) required to partition the system. We deliberately chose to exclude any automatically calculated dependencies of the annotated functions, requiring the programmer to explicitly specify each function to be placed into the trusted zone in order to prevent any unexpected behavior.

Recall the example in Section 5.2.6: if the developer annotates only function `fly` as `partition`, as shown in Figure 5.3 (a), the sub-callgraph of `fly` is `fly` → `getGPSLocation` and `fly` → `adjustDirection`. In that case, placing function `fly` in the trusted partition leads to zigzag invocations between the normal and secure worlds (Figure 5.3 (b)). If `fly` runs in OP-TEE, or in SGX configured for the minimal zigzag call (i.e., the threshold of "0"), this partitioning specification violates the zigzag rule. To fix such violations, the developer can annotate the

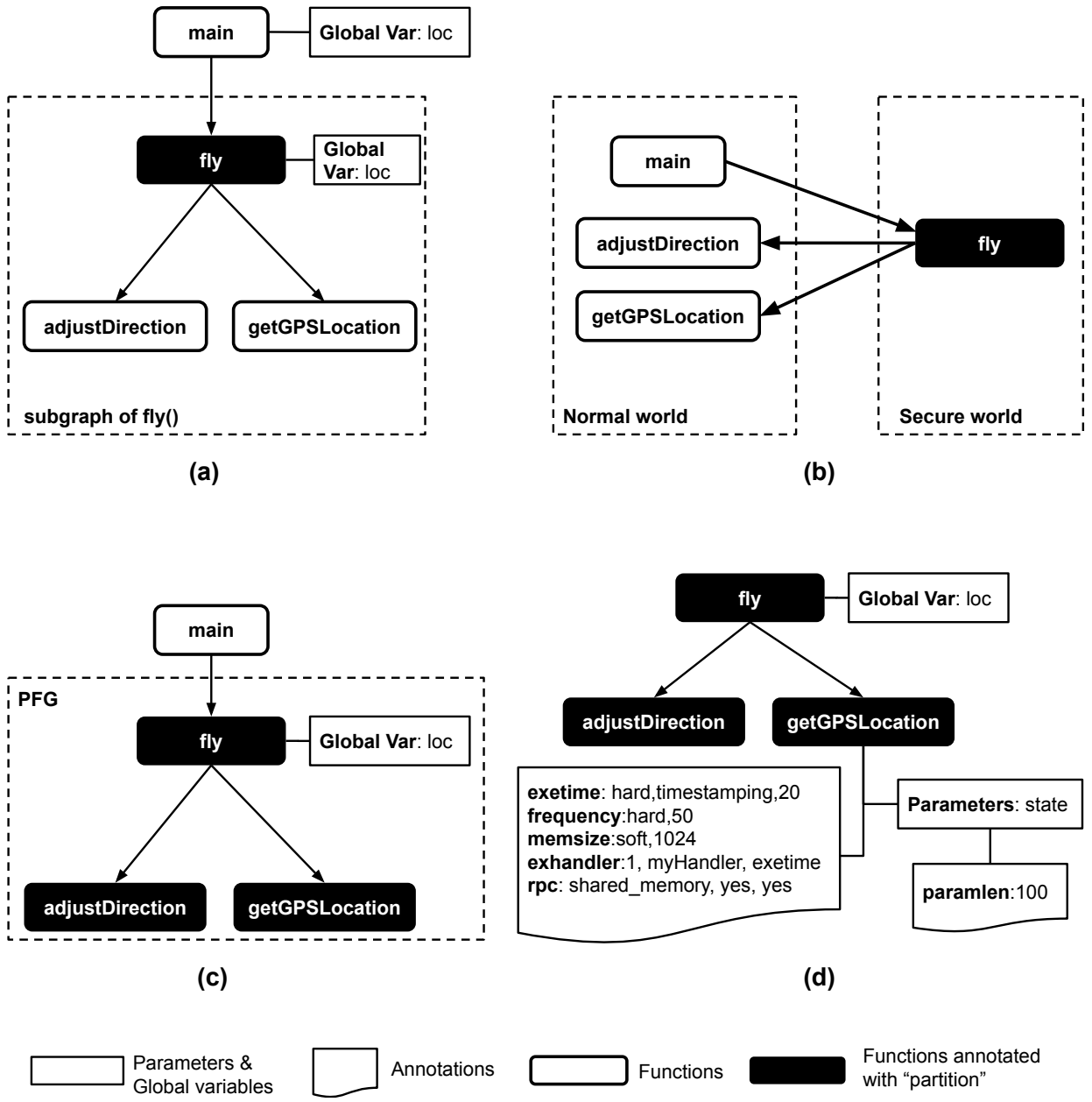

Figure 5.3: The RT-Trust PFG

other two offending functions (i.e., `getGPSLocation` and `adjustDirection`) with `partition`, so that both of them will also be placed in the secure world along with their caller `fly`. After the zigzag violation is eliminated, RT-Trust then adds `fly`'s sub-callgraph to the PFG.

Now, suppose the global variable `loc` are accessed not only by function `fly` (i.e., the secure world) but also by function `main` (i.e., the normal world). Because this scenario violates the

global variable access rule, the entire sub-callgraph of `fly` should be removed from the PFG. To fix this violation, the developer can modify function `main`, so it would no longer access `loc` (Figure 5.3 (c)), or make this global variable constant. Finally, RT-Trust constructs the PFG with all the necessary information for each function, as shown in Figure 5.3 (d).

5.3.2 Dynamic Analyses

RT-Trust offers dynamic analyses to help identify how likely the specified partitioning would meet the original real-time constraints. Since it would be hard to guarantee whether the profiled execution produces the worst-case scenario, our analyses are applicable only to soft real-time systems.

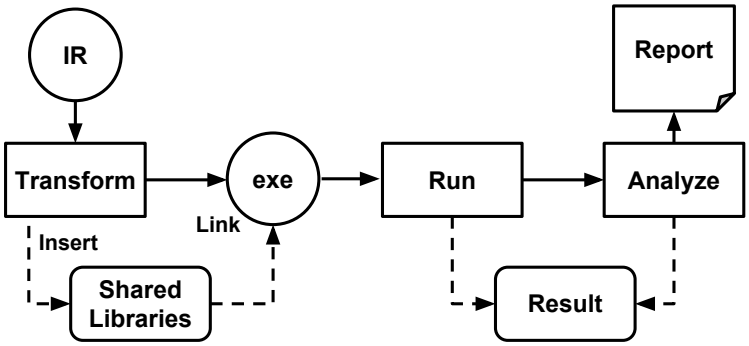

Figure 5.4: The RT-Trust Analyses Procedure

Figure 5.4 shows how RT-Trust provides the dynamic analyses capability. The analyses start with the transformation of the original LLVM IR program. That is, RT-Trust inserts profiling code at the affected call sites of the annotated functions for their corresponding real-time constraints. Instead of inlining the entire profiling code, RT-Trust inserts calls to special profiling functions, which are made available as part of shared libraries. Currently, RT-Trust provides them on its own, but similar profiling functionality can be provided by third-party libraries as well. This flexible design enables developers to provide their custom profiling libraries or add new features to the libraries provided by RT-Trust to further enhance the profiling logic. After linking these shared libraries with the transformed IR program, developers run the executable to trigger the inserted function calls to invoke the

profiling functions in the shared libraries. These functions measure the real-time constraints and persist the result data for future analysis. Finally, RT-Trust analyzes the data, estimating whether the annotated functions can meet the original real-time requirements, and reporting the results back to the developer.

Analyzing Time Constraints: As mentioned in chapter 2, time constraints mainly include the time deadline and the periodicity limit. The former defines the upper boundary for a function to complete its execution, the latter restricts the time that can elapse between any pair of invocations.

To analyze these constraints, RT-Trust first transforms the original LLVM IR program via two key steps: 1) find the correct call sites, and 2) insert the suitable function calls. In the transformation procedure below, given a function annotated with `exetime`, RT-Trust traverses its instructions to locate the first instruction in its entry basic-block[1], inserting the profiling probes and then that starts a profiling session. Likewise, RT-Trust locates each return instruction of the annotated function, inserting the probes that issue the end profiling session, which stops the profiling.

```
1 define i32 @function(i8* %param) { // annotated function
2     entry:
3      <--- start probe()
4     %first instruction
5     ...
6      <--- stop probe()
7     ret i32 %retval
8 }
```

[1]Basic-block is a straight-line code sequence. It has no *in* branches, except at the entry, and no *out* branches, except the exit.

Which probe functions are inserted depends on how RT-Trust is configured by means of RTTAs. The two main configurations are timestamping and sampling. For timestamping, RT-Trust inserts probes that invoke the timestamp functions to retrieve the current system time by means of `gettimeofday()` (in the normal world), or `TEE_GetREETime()` (in the secure world to check the adherence to real-time constraints post-partitioning). For sampling, RT-Trust inserts invocations to the sampling functions of `ProfilerStart()` and `ProfilerStop()`, which make use of gperftools (a third-party profiling tool). Similarly, to analyze periodicity limits, RT-Trust locates the first instruction of the function annotated with `period`, and then inserts invocations of the functions to record the current system time.

All these measured results are first stored in a hash table, with the key corresponding to the annotated function's name and the value to its profiling record. Finally, the hash table is persisted into an external file for further exploration.

Memory Consumption Profiling: Memory consumption is an important issue for trusted execution. First, TEEs are designed to occupy limited memory space (as discussed in chapter 2). In addition, pointer parameters of the trusted functions refer to data structures that need to be dynamically allocated as part of their marshaling/unmarshaling phases (as discussed in Section 5.2.3). To ascertain the expected memory consumption requirements of the SPI functions, RT-Trust profiles the amount of memory consumed by the functions annotated with `memsize`. The profiling comprises the traversal of the functions' IR instructions to locate all the allocation sites (i.e., the `alloca` instruction). Each allocation site is then instrumented to keep track of the total amount of allocated memory.

```
1  %var = alloca i32, align 4
2  <--- function(i32, 4)
```

The allocated memory volume is continuously monitored as the profiled system is being executed. The presence of pointers complicates the profiling procedure. To properly account for all the memory consumed by the data structure referenced by a pointer, RT-Trust implements a heuristic approach based on SoftBound [123]. To provide effective memory safety checking, SoftBound transforms the subject program to keep the base and bound information for each pointer as metadata. This metadata is passed along with the pointer. In other words, when passing the pointer as a parameter from one function to another, the metadata is also be passed. SoftBound makes use of this metadata to enforce program memory safety.

Based on SoftBound, RT-Trust inserts invocations to record the pointer metadata (base and bound) of the annotated function, whenever pointers are allocated or accepted as parameters from other functions. RT-Trust calculates each pointer's length via the formula $length = bound - base$. By combining the basic and pointer type's lengths, RT-Trust finally determines the upper boundary of the memory volume consumed by each annotated function.

5.3.3 Exception Handling

Having annotated a function with real-time constraints, developers can also specify how to handle the violation of these constraints via the `exhandler` annotation. To locate the correct call site for inserting exception handling code, RT-Trust traverses instructions of each defined function in the original program, finding the invocations to the annotated functions. Then, RT-Trust inserts "if-then-else" blocks by means of LLVM API `SplitBlockAndInsertIfThenElse`. The "if-then-else" blocks include: 1) the block that contains `if` condition, 2) "then" block, 3) "else" block, and 4) the block after "then" and "else" blocks. RT-Trust creates an `if` condition with the annotated threshold for the number of violations of a given real-time

constraint. Then, it inserts the invocation to the specified exception handling function into the "then" block, and inserts the invocation to the original function into the "else" block as follows:

```
1 Ret = function(Args); // is transforms into:
2 Ret = (t reaches threshold) ? exhandling_function(Args)
3                             : function(Args);
```

Then, RT-Trust inserts another invocation before the "if-then-else" blocks to calculate the number of observed violations of the given real-time constraint (i.e., "t" in the above code snippet). Finally, the inserted code logic can automatically switch between the original function and the exception handling function, which can be specified by the developer or generated by RT-Trust as a default option.

5.4 Inter-World Communication: Code Generation & Transformation

The partitioning process divides the program's IR into the trusted and regular parts. Our partitioning strategy is function-based: SPI-dependent functions execute in the trusted partition, while all other functions execute in the regular one. The TEE isolation mechanisms make it impossible to directly invoke SPI functions running in the trusted partition. However, each TEE provides special communication channels that can be accessed through environment-specific APIs. Hence, RT-Trust replaces the direct SPI function invocations with communication through the TEE channels for both OP-TEE and SGX.

For OP-TEE, RT-Trust first generates an RPC client stub (for the normal world) and a server stub (for the secure world). The client stub passes the function's parameters and

its unique ID, which identifies the function to execute in the secure world. The server stub receives this information and invokes the corresponding SPI function in the trusted partition. For SGX, RT-Trust generates a proxy for each SPI functions and an Enclave Definition Language (EDL) file that provides metadata for all the SPI functions. By passing the generated EDL file as input to the Edger8r tool [88], developers then generate the required SGX communication logic for all interactions between the regular and trusted parts. For both OP-TEE and SGX, RT-Trust redirects the direct invocation of a SPI function to its RPC stub (for OP-TEE) or its proxy function (for SGX).

5.4.1 Generating RPC stubs for OP-TEE

RT-Trust generates RPC stubs based on the developer's configuration in annotation `rpc` and `paramlen`. The argument `<type>` of `rpc` specifies which underlying delivery mechanism (i.e., shared memory or socket) to generate. This delivery mechanism also depends on the actual TEE implementation in place. To exchange data between the normal and secure worlds, OP-TEE provides 4 shared memory buffers, used as the delivery mechanism. However, RT-Trust must marshal/unmarshal function parameters to and from these buffers. This explicit parameter marshaling makes the generated code suitable for any communication mechanism.

The client stub includes four code sections: 1) `prologue` initializes the TEE context and opens the communication session, 2) `epilogue` closes the session and finalizes the context, 3) `marshaling` allocates memory space and marshals the function's parameters, and 4) the RPC function communicates between the normal and secure worlds by calling TEE API methods `TEEC_InvokeCommand`. Correspondingly, the server stub also includes four code sections: 1) the entry points of opening and closing the communication session, 2) `unmarshaling` unmarshals the received data, 3) a dispatcher that receives invocations and data from the client stub, and forwards it to corresponding SPI wrapper functions, and 4) the wrapper functions receive

the data from the dispatcher and invoke the actual SPI functions in the trusted partition.

During the code generation, RT-Trust checks the arguments `<encryption>` and `<compression>` of annotation `rpc`. If the developer specifies that `<encryption>` or `<compression>` is needed, RT-Trust encrypts and compresses the data after the `marshaling` phase in the client stub, and decrypts and decompresses the data before `unmarshaling` phase in the server stub. Although RT-Trust uses existing open source libraries for encryption and compression, developers can switch to using different implementations. Further, when generating the `marshaling` component for the client stub, RT-Trust checks the `paramlen` to determine how much memory to allocate.

For ease of portability, all generated code is compliant with the C language specification, without any custom extensions. Furthermore, all the referenced libraries are open source and plug-in replaceable. Finally, all the TEE APIs in the generated code conform to the Global Platform Specification of TEE. Thus, developers can either directly use the generated code for the trusted execution or extend that code in order to meet some special requirements.

5.4.2 Generating proxy functions and EDL file for SGX

Based on the partitionable functions' information in the PFG, RT-Trust generates an EDL file, assembling the declarations of trusted functions into the "trusted" block, and that of regular functions invoked from the trusted part in a zigzag pattern into the "untrusted" block. Most importantly, for each pointer parameter in both the trusted and untrusted function blocks, RT-Trust checks the `paramlen` annotation to generate the EDL attributes that determine the size of pointer-based parameters. For each function containing `struct` parameters, RT-Trust generates a complete definition of each `struct` in the EDL file. After that, RT-Trust generates a proxy function file to initialize/deallocate the communication channel and to handle the return values for each SPI function. Finally, RT-Trust executes

the Edger8r tool to generate the required SGX communication logic for this partitioning scenario.

5.4.3 Redirecting Function Calls

As SPI functions are moved to the secure world, their callers need to be redirected to invoke the original function's RPC stubs (for OP-TEE) or proxy functions (for SGX) instead. RT-Trust exhaustively examines all function invocation instructions, locates the ones invoking the SPI functions, and replaces the callee's name to the SPI function's RPC stub or proxy function. Since SPI functions and their RPC stubs / proxy functions share the same signature, no other changes are necessary:

```
1 Ret = original_function(Args); // is transformed into:
2 Ret = RPC_function(Args); // for OP−TEE
3 Ret = un_function(Args); // for SGX
```

Now, the original function calls become RPC or proxy function invocations that end up calling the partitioned SPI functions in the secure world. As per the transformation of exception handling in Section 5.3.3, the original function can be specified to handle exceptions. That is, if the violations of real-time constraints reach the threshold, the inserted exception handling logic can automatically change back to invoking the original function rather than the function in the secure world:

```
1 Ret = RPC_function(Args); //is transformed into:
2 // for OP−TEE:
3 Ret = (reach threshold) ? original_function(Args) : RPC_function(Args);
4 // for SGX:
5 Ret = (reach threshold) ? original_function(Args) : un_function(Args);
```

5.4.4 Data Encoding Protocols

The normal and secure worlds are represented by distinct system components, running in separate address spaces. The inter-process communication facility, through which the worlds interact with each other, require that all the data passed between them be encoded as an array of bytes. RT-Trust has to be able to encode the regular part's data structures into this array of bytes, while the corresponding trusted part has to read these data structures from the array once it is transferred to the secure world. This problem is not new, and multiple marshaling mechanisms [147] have been introduced, including major framework platforms, such as CORBA [159] and gRPC [76]. For SGX, the Edger8r Tool parameterized with an Enclave Definition Language (EDL) file [89] automatically generates the required marshaling/unmarshaling logic. However, OP-TEE provides no such marshaling/unmarshaling facilities. To solve this problem, RT-Trust provides a custom marshaling framework that not only generates the required marshaling/unmarshaling logic for the parameters of SPI functions, but also introduces a novel space-efficient encoding for data collections. Given that TEE is frequently used as a secure data storage, this ability to encode data collection parameters space-efficiently increases the applicability of RT-Trust.

Figure 5.5 shows how RT-Trust differently encodes parameters that are: a) primitive types (e.g., `int`, `char`, `double`), and b) complex type (e.g., `struct`, `union`). The encoding represents all data as a byte array, and when storing both primitive and complex data, it starts with the same header that contains the `total len` (the total length of all the entries in this encoding), and `num` (the total number of items in the encoded collection) fields. These fields are both stored into a 4 bytes integer. The following entries differ depending on the encoded type. For primitive types, RT-Trust then stores the size of the encoded data type, which is then followed by the actual data content. For complex types, RT-Trust first stores the type header: the `total len` (the total length of all the members in this type), and `num` (the total

number of members in this type) fields, followed by the size of each member and its actual content in turn. This scheme enables the receiving party to first extract the total length to be able to allocate the amount of memory required to contain the entire encoding. The transfer process needs to allocate memory twice: first in the shared memory, which serves as a delivery vehicle to the secure world, and then in the trusted part to be able to store the transferred data.

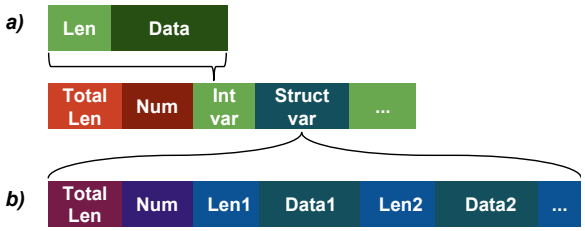

Figure 5.5: Format of Data Transmission.

5.5 Support for Partitioning Decision Making

As discussed in Section 5.2.4, for each function to partition, developers can indicate whether it must abide by `hard` or `soft` real-time constraints. Hard constraints cannot be violated, while soft ones can tolerate some violations. Hence, upon detecting a possible violation of a hard constraint, RT-Trust rejects the request to partition the offending function. For compliant SPI functions and those violating only the soft constraints, RT-Trust calculates their Function Performance Indicator (*FPI*) discussed next.

Function Performance Indicator. The Function Performance Indicator (*FPI*) reflects by how much a SPI function is expected to degrade its performance once moved to the TEE. For each appropriate SPI function, RT-Trust calculates and reports its *FPI*, upon which developers can determine whether or not to move the function to TEE. *FPI* correlates two platform-independent metrics: execution time loss (L_{exe}) and invocation interval loss

(L_{inv}). We calculate the expected performance degradation (T_{after}/T_{before}), and then scale and normalize it by applying *log* and *tanh* functions in turn[2].

Finally, we calculate the maximum value of the normalized results to obtain FPI :

$L_{exe} = T_{after}/T_{before}$; ($T_{before}$, T_{after} are execution times) (1)

$L_{inv} = I_{after}/I_{before}$; ($I_{before}$, I_{after} are invocation intervals) (2)

$FPI = Max(tanh(log(L_{exe})), tanh(log(L_{inv})))$ (3)

FPI shows the expected performance degradation factor. Notice that *FPI* can take upon values that range between 0 and 1. We offer the following guidelines to developers, as based on the ranges of *FPI* values: between 0 and .25, the expected degradation is *minimal*; between .26 and .75, the degradation is *medium*; and between .76 and 1, the degradation is *high*. Which level of performance degradation is acceptable for a given application scenario is up to the developer to determine.

For example, a SPI function `f` is annotated to be moved to TEE. Before moving `f`, its execution time and invocation interval are 1 and 5 seconds, respectively. After moving `f` to TEE, its time and interval become 10 and 20 seconds, respectively. Hence, `f`'s L_{exe} is $10/1 = 10$, L_{inv} is $20/5 = 4$, resulting in *FPI* of $Max(tanh(log10), tanh(log4)) = 0.76$. In other words, moving `f` to TEE would increase its execution costs by a factor of 0.76. This performance degradation level is in the low range of high.

As a simple but intuitive metric, *FPI* provides a convenient heuristic that can help developers determine whether moving a SPI function to the TEE would continue satisfying the timeliness requirements. Under SGX and OP-TEE, *FPI* can differ for the same SPI functions. So this metric can also help developers select the most appropriate TEE implementation for a given

[2]The *log* and *tanh* functions are classic data analysis tools. Here we apply *log* to display a large range of quantities in a small scale, and apply *tanh* to normalize the scaled result to fall within the range of 0 to 1.

real-time system.

5.6 Evaluation

We answer the following research questions in our evaluation:

- **Effort:** How much programmer effort is saved by applying RT-Trust?
- **Performance:** What is the added performance overhead imposed by performing a RT-Trust profiling on a representative real-time system?
- **Value:** How effectively can RT-Trust determine whether a planned refactoring would preserve the original real-time constraints?
- **Accuracy:** How accurately can our profiling infrastructure predict the expected performance deterioration caused by a RT-Trust refactoring?
- **Limitations:** What are some limitations of RT-Trust's applicability?

5.6.1 Experimental Setup

To answer the evaluation questions above, we have concretely implemented RT-Trust and assessed its various characteristics in a realistic deployment scenario, whose experimental setup is as follows.

Software and Hardware: RT-Trust integrates RTTAs with the public release of Clang 4.0 and implements a series of LLVM Passes (e.g., code analysis, partition, RPC stubs generation, profiling code insertion) in LLVM 4.0. Since our memory consumption profiler relies

on SoftBound, which runs only in LLVM 3.4, RT-Trust implements a separate LLVM Pass that profiles the memory consumed by specified functions in that earlier LLVM version. For OP-TEE, the benchmarks that we use for evaluating RT-Trust are set up on Raspberry Pi 3 (RPi3), running OP-TEE 3.1.0 on Linux version 4.6.3, 1.4GHz 64-bit quad-core ARMv8 CPU, and 1 GB SDRAM. For SGX, the evaluation environment are set up on a Dell workstation, running Intel SGX Linux 2.0 Release on Ubuntu 16.04, 3.60GHz 8-core Intel i7-7700 CPU, with 31.2 GB memory.

Microbenchmarks and Realistic real-time system: Real-time systems that can benefit from RT-Trust possess two characteristics: 1) have SPI-dependent functions that should be protected in the secure world, and 2) have the execution of these functions restricted by some real-time constraints.

To establish the baseline for the performance behavior of such systems, we choose several classic algorithms as our microbenchmarks, which are widely used by existing real-time system. To mimic the real-time invocations of our microbenchmarks, we have written custom unit test suites that exercise the SPI-dependent functionality. For example, we simulate the invocation of a certain algorithm 50 times. The selected benchmarks are algorithmic in nature and include CRC32, DES, RC4, PC1, and MD5. One can imagine realistic application scenarios, in which the execution of these benchmarks needs to be protected under real-time constraints. Because both OP-TEE and SGX support only C code as running in the secure world, we select the C implementations of these algorithms provided by one of the LLVM test suites [9].

To ascertain the applicability of RT-Trust to an actual real-time system, we apply it to secure two SPI tasks of an open-source autopilot PX4 (v1.8.0) [131]: airspeed and waypoint computations.

Evaluation Design: As described in Section 5.3 and 5.4, developers can customize the implementations of profiling, EDL file and RPC stubs. However, we evaluate only the default options of using RT-Trust to establish its baseline performance, thus not unfairly benefiting our implementation.

We evaluate programmer effort as the uncommented lines of code (ULOC): 1) those required to write RTTAs, 2) those automatically generated by RT-Trust, and 3) those that the developer is expected to fine-tune by hand (e.g., some source code may need to be modified to fix the violations of our partitioning rules, or the parameter's length in an RPC stub / EDL file may need to be manually adjusted). Note that RT-Trust generates tight code, without any redundancies or unnecessary features, very similar to what a programmer would write by hand. Hence, we argue that without RT-Trust, programmers would be writing all the generated code by hand. By reporting on the size of this code, we measure how much programmer effort RT-Trust saves.

To evaluate performance, we measure the overhead of RT-Trust's profiling for execution time, invocation interval, and memory consumption. For the former two, RT-Trust provides different profiling libraries, applying TEE (i.e., OP-TEE or SGX) APIs in the secure world. So we evaluate them in both the normal and secure worlds. For the latter, memory consumption should be profiled before partitioning and generating RPC stubs or the EDL file. So, we evaluate it only in the normal world.

To evaluate value and accuracy, we first apply RT-Trust to profile the specified SPI functions before and after moving them to the secure world. Then, we compare the results reported by the profiling of the original unpartitioned system with respect to meeting the real-time constraints with that of its partitioned version.

However, the time measurement's granularity in the OP-TEE time API differs from that

in the SGX API, which reports the time-elapsed quantities only at the seconds level of granularity. To effectively measure the SPI functions' performance (at the milliseconds level) under SGX, we modified the source code to repeat each benchmark 1000 times. Despite these repeats, we report the final results at the millisecond level of granularity by simply dividing them by 1000. By using the same measurement unit for both OP-TEE and SGX, our experimental results provide a realistic comparison of the expected performance degradation levels imposed by these TEE implementations. Also, by using *FPI*, developers can effectively compare the performance of a given SPI function in different TEE implementations.

Further, by analyzing the performance results, we discuss 1) which procedure causes the performance deterioration after moving the SPI function to the secure world, 2) whether we can accurately predict the specified function's performance in the secure world by analyzing its performance in the normal world, and 3) which TEE implementation can better preserve the timeliness requirements of our evaluation cases. To explain RT-Trust's limitations by describing several program cases that require a prohibitively high programmer effort to adjust the generated RPC stubs.

5.6.2 Results

We verify the correctness of RT-Trust by applying all its LLVM passes (i.e., code analysis, transformation, and generation) to microbenchmarks. We evaluate RT-Trust as follows.

Effort: Table 6.5 shows the effort saved by applying RT-Trust. Generally, the total number of ULOC automatically generated and transformed by RT-Trust ($244 \sim 388$ ULOC for OP-TEE; $46 \sim 87$ ULOC for SGX) greatly surpasses those required to manually annotate (< 5 ULOC) and modify ($0 \sim 15$ ULOC) the subject programs.

Table 5.1: Programmer Effort (ULOC)

Algorithm	RTTAs	Generate & Transform		Adjust	
		OP-TEE	SGX	OP-TEE	SGX
CRC32	5	388	87	0	0
PC1	4	344	73	6	6
RC4	3	292	61	3	1
MD5	3	364	86	3	1
DES	2	244	46	15	3

RT-Trust eliminates the need for the developer to write this code. In other words, to apply RT-Trust, the developer adds a tiny number of ULOC, mainly as annotations and minor adjustments of generated code. The number of annotations is directly proportional to the number of SPI functions. The manual adaptations are required to remove program patterns that prevent RT-Trust from successfully partitioning the code, and to support the pointer parameters of SPI functions.

Specifically, to move the 5 SPI functions of CRC32 to the secure world requires exactly 5 ULOC of RTTAs. No manual adjustment is necessary, as the code comes amenable to partitioning and no pointer parameters are used. In contrast, 15 (for OP-TEE) and 3 ULOC (for SGX) are required to adjust the generated RPC communication for DES, due to a SPI function's pointer parameter pointing to a `struct` of two `char` arrays. In other words, after profiling the amount of consumed memory, the developer needs to adjust the memory allocation for marshaling/unmarshaling these pointer parameters. For PC1, 6 additional ULOC are needed to fix a violated global variable rule.

Overall, the number of generated and adjusted lines of code needed for SGX is generally fewer than those for OP-TEE. The reason is that, for SGX, RT-Trust only needs to generate an EDL file to construct the communication channel, while the developer only needs to modify the size or count modifiers in the EDL file to adjust the amount of memory allocated

for the pointer parameters.

Table 5.2: Overhead of RT-Trust profiling (ms)

Algorithm	Execution Time		Invocation Intervals		Memory	
	Normal	Secure	Normal	Secure	Parameter	Local
OP-TEE	0.442	144	0.418	139	0.051	0.053
SGX	0.2	52	0.212	25.5		

Performance: Table 5.2 reports on the overhead of RT-Trust profiling, which captures and calculates the execution time, invocation intervals, and memory consumption. Recall that RT-Trust profiles systems *before* and *after* refactoring them. The *before* mode estimates whether the refactored system would continue meeting real-time constraints, while the *after* mode compares the estimated execution characteristics with those performed on TEE hardware (OP-TEE on a Raspberry Pi3 and SGX on a Dell workstation). Hardware environments heavily impact the profiling overhead, with an order of magnitude difference: for OP-TEE, $\approx 0.4ms$ in the normal world vs. $\approx 140ms$ in the secure world. For SGX, $\approx 0.2ms$ in the normal world vs. $\approx 50ms$ in the secure world. This drastic performance difference is mainly due to the differences between the efficiency of standard Linux system calls and their TEE counterparts. For example, the standard `gettimeofday` is more efficient than either `TEE_GetREETime` in the OP-TEE or `sgx_get_trusted_time` in the SGX.

The heavy performance overhead of trusted execution prevents the profiling of real trusted system operation. When estimating memory consumption, the overhead of capturing the memory allocated for local variables and the pointer parameters never exceeds $0.06ms$. However, the overall overhead depends on the total number of local variables and pointer parameters. For example, if a function allocates memory for n variables, the total overhead would be $\approx 0.053 * n$ (ms). Thus, to prevent the profiling overheads from affecting the real-time constraints, the RT-Trust profiling is best combined with the system's testing phase.

Table 5.3: Value and Accuracy of RT-Trust (ms)

Alg.	Comm.		Execution Time				Invocation Interval				Memory (bytes)	
			Before		After		Before		After		Parameter	Local
CRC32	253.17	28.91	1.150	0.21	1.3	≈ 0.0	1.240	0.23	269	29.15	40	92
PC1	273.38	29.03	68.22	7.64	13	8.95	68.10	8.10	314	37.67	32	22
RC4	236.96	29.62	500.52	32.89	447	66.00	506.95	33.19	705	97.10	240	1144
MD5	177.83	30.90	267.43	49.72	254	49.35	267.62	51.08	446	78.71	20000	316
DES	201.99	28.84	24.18	2.51	32	3.18	24.30	2.55	224	31.56	528	72
airspeed	256.35	32.87	≈ 0.0	0.01	≈ 0.0	≈ 0.0	50.16	53.75	305.0	83.29	12	12
waypoint	264.96	32.38	0.400	0.05	0.460	≈ 0.0	500.75	505.98	773.67	533.32	40	40

Value and Accuracy: Table 5.3 shows the results of profiling the SPI functions, with the profiling overhead subtracted. The value before "$\vert$" is the results for the OP-TEE, and after "$\vert$" is that for the SGX. For the execution time, generally, the time consumed by our micro-benchmarks and the SPI PX4 functions in the secure world ("After" column) is similar to that in the normal world ("Before" column). Hence, moving the SPI functions to TEE should not deteriorate their performance. Thus, it is reasonable to estimate the performance in the secure world based on that in the normal world. However, the communication channel between the normal and secure worlds slows down the invoked functions due to the introduction of two time-consuming mechanisms: connection maintenance to the secure world (e.g., initialize/finalize context, open/close session), and invoking the partitioned functions in the secure world (e.g., allocate/release shared memory, marshal and unmarshal parameters).

Given a real-time deadline to complete the execution of a SPI function, the post-refactoring profiling helps determine if the deadline is being met. The source code for PX4's airspeed calculation sets the *execution timeout* to 300 milliseconds. Since the maximum post-refactoring latency of 256.35 (in OP-TEE) is below this deadline, moving this SPI function to TEE preserves its real-time constraints.

The time spent in the communication channel increases the invocation intervals of our

micro-benchmarks and the SPI PX4 functions. The micro-benchmarks invoke functions consecutively in a loop. Thus, in the normal world, each function's invocation interval ("Before" column of "Invocation Interval") is similar to its execution time ("Before" column of "Execution Time"). However, in the secure world, these invocation intervals increase, becoming similar to the time consumed by Communication ("Communication" column) plus the time in the secure world ("After" column of "Execution Time"). For the PX4 autopilot, which computes the airspeed and next waypoint values every 50ms and 500ms, respectively, the time spent in the communication channel increases these invocation intervals to 305ms ($\approx 256.35(communication) + 0$(execution time) $+50$) and 773.67ms ($\approx 264.96(communication) + 0.46$ (execution time) $+500$) in OP-TEE, and to 83.29ms ($\approx 32.87(communication) + 0$(execution time) $+50$) and 553.32ms ($\approx 32.38(communication) + 0$(execution time) $+500$) in SGX. Hence, the introduced remote communication between the normal and secure worlds is the performance bottleneck of trusted execution.

The memory consumption profiling helps determine which functions can be run in the secure world. Based on the profiled memory consumed, developers can increase the size of TEE's shared memory. For example, if the TEE's memory size is limited to $10 * 1024$ bytes, and the MD5's `char` pointer parameter requires 20000 bytes, to run MD5 in the secure world requires modifying the TEE hardware configuration. The PX4 SPI functions (i.e., airspeed and next_waypoint), which perform numeric computations, require limited memory (i.e., for the `double` / `float` parameters / variables).

5.7 Discussion

In this section, we first discuss the limitations of TEE implementations and RT-Trust. Then after comparing the OP-TEE with the SGX, we discuss their most suitable usage scenarios.

5.7.1 Limitations

TEE Limitations: Table 5.4 shows the limitations of the OP-TEE and the SGX. For language support, the trusted part for the OP-TEE can only be written in C; that for the SGX can be written in both C and C++, while the communication channel between the trusted and untrusted parts can be written only in C. For memory allocation, the OP-TEE has no fixed size limit, with the upper bound becoming the amount of physical memory. In contrast, the maximum size of the SGX's protected memory is limited by the system BIOS with 64MB or 128MB as the typical value. Besides, neither the OP-TEE nor the SGX provides any support for multi-threading in the secure world. That is, one cannot spawn a thread (e.g., by using `pthreads`) inside the secure world. Furthermore, both TEEs re-implement their special versions of the standard system and C/C++ libraries. For example, the `printf` implementation of the OP-TEE cannot print `float` or `double` values. Similarly, the SGX provides neither `strcpy` nor `strcat`, instead requiring that developers use the provided `strncpy` and `strncat` instead [91].

Table 5.4: TEE Limitations

Limitations	OP-TEE	SGX
Language	C	C/C++
Memory	no limit	hard limit
Threading	no	no
Sys./lang. APIs	special version	special version

RT-Trust Limitations: For OP-TEE, consider the scenario of passing a struct pointer to the specified function. The struct pointer is a linked list that has 100 elements. Each element has a `char` pointer as the data field. In that case, developers need to modify more than 100 ULOC in the generated RPC stubs to allocate the correct memory size for the marshaling and unmarshaling operations. In other words, the more complex pointer-based data structures are, the greater the programming effort is required to adapt generated code. Thus, the utility

of RT-Trust diminishes rapidly for refactoring functions with complex pointer parameters.

For the SGX, RT-Trust requires that developers write specialized logic to marshal/unmarshal such complex pointer parameters. If the size of a pointer-based parameter happens to be larger than the limit set by the system BIOS, developers need to do extra work. First, modify the source code to divide the parameter data into several smaller parts and then write the required code to marshal/unmarshal the divided data to be transferred across the normal and secure worlds.

For both OP-TEE and SGX, RT-Trust restricts SPI functions from having function pointer parameters. Further, RT-Trust rejects the refactoring requests in which a SPI function assigns function pointers within its body. By inspecting the `AllocaInst` instructions during the static analysis phase, RT-Trust locates function pointers in the bodies of SPI functions. Upon detecting the presence of a function pointer, RT-Trust raises a partition failure. Besides, sometimes dynamically allocated objects can significantly differ in size depending on input. Hence, systems must be profiled with typical input parameters.

5.7.2 Choosing between OP-TEE or SGX

Table 5.5: FPI of OP-TEE and SGX

Algorithm	OP-TEE	SGX
CRC32	0.982	0.973
PC1	0.581	0.6
RC4	0.142	0.435
MD5	0.218	0.205
DES	0.756	0.803

Table 5.5 shows each micro-benchmark's Function Performance Indicator (FPI) for the OP-TEE and the SGX. Overall, the FPI values are comparable for both TEEs in all benchmarks. The faster the execution before moving to the TEE, the larger the FPI value (i.e., more

performance degradation). The reason is that if a function runs fast (e.g., 1.15 ms for CRC32), the additional costs of the communication channel (i.e., 253.17 ms for CRC 32) dominate the total execution time. Another concern is the execution latencies in the secure world. In the case of RC4, moving the SPI functions to the SGX doubles their execution time. However, after moving the same functions to the OP-TEE, the execution time stays similar (as shown in Table 5.3). Hence, RC4's FPI for the SGX (i.e., 0.435) is larger than that for the OP-TEE (i.e., 0.142). To sum up, developers should always use the TEE with the smallest FPI value. However, if a SPI function's execution time is much smaller than the time taken by the communication channel, then both the OP-TEE and the SGX impose a comparable high-performance degradation.

5.8 Conclusion

In this chapter, we have presented RT-Trust that provides a fully declarative meta-programming model with RTTA, static and dynamic analyses for determining whether the suggested partitioning strategy is reasonable, and whether the partitioned system would comply with the original real-time constraints, and an automated refactoring that transforms the original system while generating custom RPC communication and exception handling code. Our approach automatically refactors real-time systems with SPI-dependent functions for trusted execution under real-time constraints. The evaluation results of applying RT-Trust to micro-benchmarks and a drone autopilot indicate the promise of declarative meta-programming as a means of reducing the programmer effort required to isolate SPI under real-time constraints.

Chapter 6

Identifying and Migrating Non-sensitive Code in TEE

A soaring number of computing devices continuously collect massive amounts of data (e.g., biometric ids, geolocations, and images), much of which is sensitive [160]. Sensitive data and code processing them are the target of many data disclosure and code tampering attacks [11, 12, 13, 14, 48, 49]. An increasingly popular protection mechanism isolates sensitive code and data from the outside world[1] in a trusted execution environment (TEE) (e.g., SGX [47] and OP-TEE [128]). 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 [108]. For example, numerous function invocations, entering or leaving TEE, trigger a large volume of in/out communication, slowing down the entire system [163]. When it comes to security, prior works [103, 106, 108, 136, 141, 173] move programmer-specified data or functions, even the entire system (e.g., Graphene [153], Haven [32], and SCONE [29]) 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 [122], any vulnerable or malicious functions inside TEE can compromise the

[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 chapter 2-2.1.2).

security of the entire system. For example, memory corruption attacks can exploit vulnerabilities within a TEE-based function[2] [33, 102]. 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;

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

(b) modifies the system's intermediate representation (IR) to move the developer-confirmed *non-sensitive functions* to the outside world.

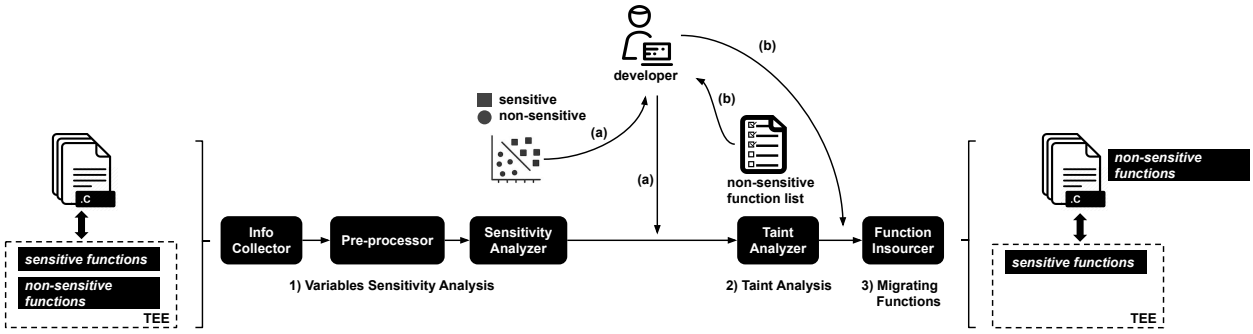

Figure 6.1: TEE-DRUP overview

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 work 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.

6.1 Solution Overview

In this Section, we explain TEE-DRUP's application by example.

6.1.1 TEE-DRUP Process

Figure 6.1 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 re-

ported 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 TEE metadata[3], 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.

```
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     char * error_des = malloc(100 * sizeof(char));
21     sgx_status_t status = log_erros ( global_eid , error_des );
22
23     /* Destroy the enclave */
24     sgx_destroy_enclave (global_eid);
25
26     ...
27
28     return 0;
29 }
```

```
1  #include "Enclave_t.h"
2
3  int get_airspeed() { ... }
4
5  char* log_erros() { ... }
```

Figure 6.2: Example code

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

6.1.2 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 [92], with the side-by-side code snippets appearing in Figure 6.2. 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[4], 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

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

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[5].

6.2 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).

6.2.1 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 6.1 shows an encoded record for variable `const int * the_password`.

[5]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.

Table 6.1: Data Format: `const int * the_password;`

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

6.2.2 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.

6.2.3 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 four rationales. The first three rationales has been discussed in chapter 4-4.4, here we introduce the last one:

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. Our evaluation presented in Chapter 4 shows that, in general, 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.

2) Sensitivity Computation Algorithm. Algorithm-2'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).

Algorithm 2 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

Function: get_similarity(word_array, *dict*, λ):
$sim \leftarrow 0$
foreach *word : word_array* **do**
    $txt \leftarrow find\_most\_similar(word, dict)$
    $d \leftarrow similarity(word, txt)$
    increase sim by $d * \lambda$
end
$avg \leftarrow average$(sim)
return avg
Function: calculating_main(textual_info_list, *dict*):
foreach *var : textual_info_list* **do**
    `/* for variable name. */`
    $var\_name \leftarrow get\_var\_name(var)$
    $sim\_var \leftarrow get\_similarity$(var_name,dict, 1)
    `/* for type name. */`
    $type\_name \leftarrow get\_type\_name(var)$
    $sim\_type \leftarrow get\_similarity$(type_name,dict, 1)
    `/* for function name. */`
    $func\_name \leftarrow get\_func\_name(var)$
    $sim\_func \leftarrow get\_similarity$(func_name,dict, 1)
    `/* for file path. */`
    $path \leftarrow get\_path(var)$
    $sim\_path \leftarrow get\_similarity$(path,dict, 0.8)
    $var.sensitivity \leftarrow$ (sim_var + sim_func $*$ 0.8 + sim_type $*$ 0.8 + sim_path $*$ 0.5)
end

3) An Example: Following the example in 6.1.2, 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 "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 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 [74]. 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 [139]. 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 6.4.

6.2.4 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 [137]. 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.

6.3 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* [150] and GNU style [69].

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.

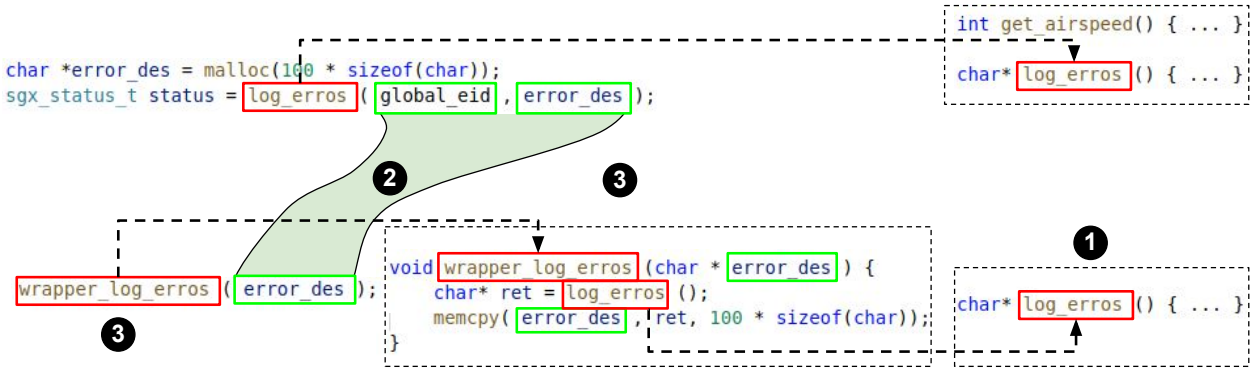

Figure 6.3: The TEE Insourcing Refactoring

Figure 6.3 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 reconstructed 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.

6.4 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?

6.4.1 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 Re-

lease. To calculate the word semantic similarity, we use Google's official pre-trained model (Google News corpus [75]). 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 6.2).

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 [31, 109]. 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 6.2: Projects Information

Project	Domain	Code-base Info		Variable Number		
		file num	LoC	total	pre-proc	no tests
GPS Tracker [16]	GPS	76	62746	1993	1298	NA
PAM module [18]	Authentication	4	709	66	28	NA
su-exec [20]	Privileges	1	109	16	13	NA
mkinitcpio-ykfde [15]	Encryption	3	1107	88	79	NA
Spritz Library [19]	Encryption	1	614	138	126	NA
libomron [17]	Health Care	7	1544	166	150	128
ssniper [149]	Personal Info (SSN)	12	2421	618	285	253
emv-tools [30]	Bank & Credit	37	9684	1104	995	862

6.4.2 Evaluation Design

1) Correctness: As shown in Table 6.2, from each project, we extracted all of its variables[6], 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 6.2.4), 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[7] among the TEE-DRUP-designated variables (non-designated vari-

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

[7]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."

ables 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.

6.4.3 Results

1) Correctness: Table 6.3 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, 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 sensitiv-

ity scores as non-sensitive. Hence, a large p-tile may lead to high false positives/negatives, lowering accuracy, precision, and recall.

Table 6.3: Correctness

Project	P-tile	Accuracy	Precision	Recall	Unsure Rate		Miss Rate
					designated vars	all vars	
	10%	90.6%	93.8%	96.2%	67.3%		70.8%
	30%	88.1%	93.0%	94.0%	74.0%	77.2%	33.1%
GPS Tracker	50%	84.1%	90.3%	91.4%	77.2%		8.6%
	10%	100%	100%	100%	0%		78.6%
	30%	92.3%	100%	88.9%	18.8%	21.4%	42.9%
PAM module	50%	72.7%	83.3%	71.4%	21.4%		28.6%
	10%	100%	100%	100%	0%		66.6%
	30%	100%	100%	100%	12.5%	30.8%	0%
su-exec	50%	88.9%	75%	100%	30.8%		0%
	10%	100%	100%	100%	50.0%		61.1%
	30%	90.0%	94.1%	94.1%	58.3%	64.6%	11.1%
mkinitcpio-ykfde	50%	78.6%	80.0%	88.8%	64.6%		11.1%
	10%	100%	100%	100%	50.0%		74.5%
	30%	97.3%	96.8%	100%	51.3%	46.8%	41.2%
Spritz Library	50%	82.0%	93.3%	82.4%	46.8%		17.6%
	10%	64.7%	57.1%	100%	43.3%		69.2%
	30%	61.7%	48.6%	100%	47.8%	52.0%	34.6%
libomron (with tests)	50%	66.7%	52.1%	96.2%	52.0%		3.8%
	10%	93.3%	91.7%	100%	42.3%		57.7%
	30%	93.8%	91.3%	100%	57.9%	60.9%	19.2%
libomron (no tests)	50%	82.0%	75.8%	96.2%	60.9%		3.8%
	10%	84.8%	79.2%	100%	43.1%		77.6%
	30%	75.9%	85.4%	77.4%	54.1%	51.6%	51.8%
ssniper (with tests)	50%	62.3%	70.9%	65.9%	51.6%		34.1%
	10%	100%	100%	100%	42.0%		76.5%
	30%	85.1%	100%	80.0%	55.9%	58.1%	52.9%
ssniper (no tests)	50%	72.6%	96.7%	68.2%	58.1%		31.8%
	10%	85.5%	92.0%	88.5%	65.5%		69.3%
	30%	72.0%	72.4%	76.7%	63.5%	61.8%	40.7%
emv-tools (with tests)	50%	65.5%	54.4%	78.7%	61.8%		21.3%
	10%	91.5%	100%	89.6%	65.7%		71.3%
	30%	79.2%	95.6%	76.3%	71.2%	71.3%	42.0%
emv-tools (no tests)	50%	70.4%	74.2%	78.6%	71.3%		21.3%

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 6.3). 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 6.4 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 6.5 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

Table 6.4: 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

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 6.5: Programming Effort (ULOC)

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

6.4.4 Discussion

1) Correctness: Based on our results (Table 6.3), 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 6.4.2-1), causing a low miss rate but a high number of false positives/negatives and low accuracy/precision/recall (row "50%" in Table 6.3). 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.

6.5 Conclusion

In this chaper, 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.

Chapter 7

Exchanging Sensitive Data

Mobile, IoT, and wearable devices continuously collect increasing volumes of user data, much of it sensitive. Health monitors track their owners' vital signs; smart phones read sensory personal data, including GPS location, velocity, direction, etc.; IoT devices obtain their environmental information. When it comes to sensitive data, there is a fundamental conflict between end-user requirements and application developer aspirations. End-users want to make sure that their sensitive data remains private, inaccessible to non-trusted parties. Application developers want to leverage the sensitive data's properties to provide intelligent applications that provide personalized user experiences and intelligent, context-sensitive services. These two objectives are seemingly irreconcilable.

Consider the following three examples of data-intensive applications that handle potentially sensitive data.

(1) Mobile navigation applications provide real-time traffic information to their users, who also contribute this information when using the applications. When providing navigation services, a navigation application continuously uploads to the cloud its device's current GPS location and speed, which are then used to estimate real-time traffic information. Although this information is uploaded anonymously, given enough GPS data and speed, a non-trusted party may be able to learn about the device owner's daily routine and abuse this information for nefarious purposes. Therefore, although the contributors may be willing to help estimate real-time traffic information, they would not want their GPS location revealed and recorded.

(2) A group of friends is looking for a restaurant to dine together. Their smartphones maintain their owners' dining histories. Based on the dining history of each individual, the edge server at a shopping plaza can suggest which restaurants would be most suitable for the entire group. However, the individuals may be unwilling to share their raw dining histories with a non-trusted party.

(3) A smart building may adjust its temperatures and lighting levels, as driven by the preferences of its current occupants. Users wearing personal health trackers and equipped with smartphones can report their owners' vitals and building location. This information is one of the key pieces that make a building "smart." However, individuals may not want to reveal their vitals and current location to a non-trusted party.

Please, notice that in all three examples above, intelligent services can still be provided while keeping the users' sensitive data invisible. To calculate the average reported velocity, the traffic monitoring system should not need to be aware of the actual velocity of each passing vehicle. This sensitive information can remain hidden, and only its statistical average calculated and used in predicting current traffic conditions. To recommend a restaurant, the edge server should not need to know which specific restaurants the individuals involved have patronized in the past. Instead, it can suggest a mutually acceptable restaurant, based on each individual's most favored cuisine, statistical information that can be obtained by querying the dining histories. To intelligently adjust its environmental settings, a smart building can remain unaware of the building inhabitants' actual vitals or exact locations. It only needs the vital signs' statistical averages, as reported by each of its autonomous climate-controlled areas.

We posit that when developing data-intensive, privacy-preserving applications, developers often need to keep the sensitive data invisible, while being able to query that data for various properties. Both data privacy and the ability to query the data should be guaranteed by the

runtime system. However, existing programming mechanisms provide no explicit support for this type of privacy preservation. In Object-Oriented programming, `private` fields can only be accessed by the `public` methods in the same class. These methods, however, are free to implement any access policy to sensitive data and even return the values of privately declared fields to the client. In addition, this method-level protection can be bypassed via Reflection [59]. If a programming mechanism can guarantee that the sensitive data will remain invisible, while providing a controlled way to query the data, this mechanism can support the development of emerging applications in the mobile and IoT domains, thereby improving their data privacy.

In this chapter, we present Object Expiration (ObEx), a novel programming mechanism, supported by a lightweight, portable runtime system, that possesses all the privacy preservation properties described above. The key idea behind ObEx objects is that they securely store some sensitive data, to which they never provide access to their clients. In other words, the sensitive data remains hidden in the system layer and never revealed to the application that uses these objects. Although the sensitive data remains invisible, clients can execute various pre-defined queries (e.g., testing for equality, comparisons, membership in a range, etc.) against it. However, the total number of queries and the time limit over which the queries can be executed are limited by a declarative policy specified when instantiating ObEx objects. The runtime system ensures that the specified policy is preserved when ObEx objects are serialized and transferred across the network. When deep-copying one ObEx object to another by using serialization, the runtime keeps only one version of the sensitive data, causing the copies to become aliases to the same data. When transferring an ObEx object across the network, the runtime system encrypts its sensitive data, and starts the expiration timer as soon as the object is unserialized at the destination site.

This work makes the following contributions:

1. We introduce ObEx objects, a programming mechanism that supports the development of data-driven, privacy-preserving distributed applications, common in the emerging mobile, wearable, and IoT domains.

2. We empirically evaluate ObEx objects in terms of their usability and efficiency with benchmarks and privacy threats.

3. We present developer guidelines for using ObEx objects in a given application scenario, as informed by our empirical observations.

7.1 Usage Scenario and Use Case

In this section, we first present a typical ObEx usage scenario. Then, we show how using ObEx can effectively reconcile conflicting requirements of data producers and consumers in a realistic use case.

7.1.1 Typical Usage Scenario

Consider a scenario of collecting and making use of some sensitive data. The data is sensitive in the sense that it contains some private information that should not be revealed to outside parties. The role of *a data producer* is to collect or generate some sensitive data. The role of *a data consumer* is to make use of the sensitive data's properties (without having the ability to access the data) to provide some intelligent services. As owners of sensitive data, producers also determine which access policy should be applied to the data, so as to preserve user privacy while permitting consumers to leverage some properties of the sensitive data.

Data producers and data consumers operate in an environment of *mutual distrust*. Producers

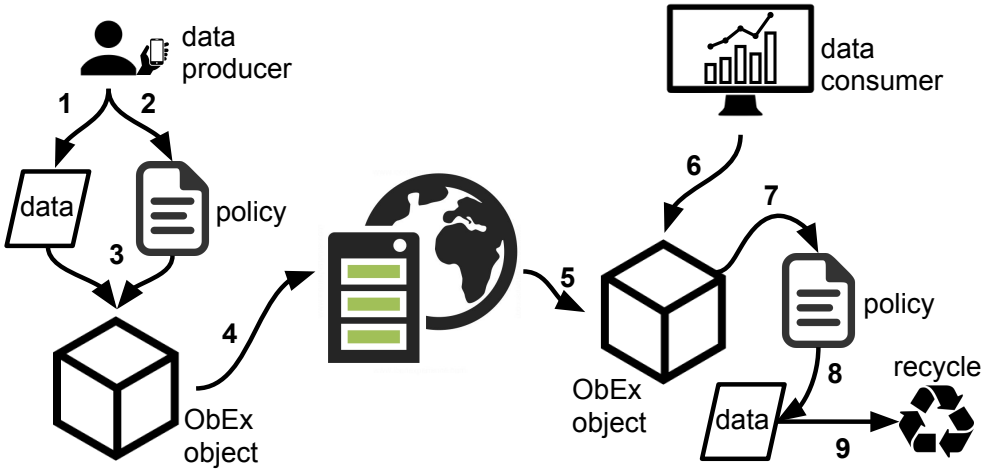

Figure 7.1: Typical Usage Scenario.

would not share sensitive data with consumers, both to preserve user privacy and to comply with privacy regulations. Consumers would not trust the results of any operations performed by producers over the data. Consumers need to be able to compute over the sensitive data on their own, as their computational procedures constitute their intellectual property (IP), which is not to be revealed to the producers.

Figure 7.1 shows a process of developing a distributed data-intensive application that preserves data privacy by means of ObEx. The process starts with the introduction of sensitive data into the system (step 1). The data is then stored in an ObEx object. When instantiating the object, the access policy determined by the producer (step 2) is passed as a parameter to the constructor (step 3). Then, the ObEx runtime system serializes, encrypts, and transfers the ObEx object to the data consumer, which is a non-trusted party that should be unable to access the sensitive data (step 4). Serializing data into a binary string, Step 4 can be easily embedded into the data marshaling/unmarshaling processes of standard web services. When the ObEx object arrives to the non-trusted party's site, the ObEx runtime decrypts and unserializes the transferred object, making it ready for client queries (step 5). The access policy dictates the type of queries, their number, and the total time over which they can be

made. The runtime enforces the policy by keeping track of the queries made and the time elapsed (step 7). Once either of the thresholds is reached, the runtime reliably clears the sensitive data associated with the ObEx object (step 8 and 9). Once the data is cleared, all subsequent queries result in a runtime exception, raised by the ObEx runtime.

Applicability: ObEx enhances the built-in language protection facilities to secure data privacy by controlling how sensitive data is accessed[1]. As such, ObEx integrates naturally with other security schemes (e.g., network secure protocols) to further protect data privacy. ObEx requires that: (1) Both the operating system and the JVM are trusted; (2) The intelligent service providers are trusted; providers offering the ObEx API themselves are non-malicious.

7.1.2 Use Case

A recent sociological study analyzed the integration and behavioral patterns of people who have recently moved to live in the city of Shanghai [172]. For the study, China Telecom supplied anonymized metadata from 698 million of traces of all its Shanghai customers. In this demographic study, researchers analyzed the provided call traces and concluded that city "locals" and "migrants" behaved dissimilarly with respect to their daily itineraries and calling patterns. It was surprising how many behavioral patterns the researchers were able to infer, given that the provided metadata only included the age, sex, and call traces of the phone customers. For example, while the locals tend to communicate with people of their own age, the migrants tend to communicate with people of various ages. As a result, by analyzing the age of one's contacts in the trace, one can determine whether the subject is likely to be a migrant.

[1] **data access** refers to *read access*, as non-trusted clients cannot update data

The study also raised some questions from civil libertarians, which worry that by disclosing even anonymized data about phone customers, phone companies enable third parties to learn sensitive information about people's lives, thus violating their privacy. Please, notice, however, that the researchers, which were given access to the anonymized metadata, can be considered *a trusted party*, someone who would not exploit the sensitive data for nefarious purposes. For these demographic researchers, the sensitive data is only a tool that enables them to infer the integration patterns of migrants moving to a new city.

However, commercial entities could leverage the age and sex information to create useful and intelligent applications and services, but additional care must be taken to preserve user privacy. Companies developing IoT and mobile applications can target certain demographics, in which "possible migrants" would be a large group with distinct interests and needs. For example, the mobile ads delivered to this group can prominently feature 'apartments for rent' information. Similarly, newcomers would appreciate more detailed guidance from navigation applications.

How can one enable software developers to leverage the sensitive data of the phone company's customers, while ensuring that the customers' privacy is fully preserved? In other words, we want developers to be able to infer which customers are *likely migrants*, without being able to infer their other behavioral patterns. Next, we show how using ObEx can reconcile the objectives of creating intelligent mobile services and preserving user privacy.

As in the typical scenario described above, the telecoms operator is the data producer, while the third-party services for the mobile devices of phone customers are data consumers. These services can provide customized business intelligence to the consumers' mobile/IoT applications. To realize this requirement, the data consumers should be able to analyze the age of the contacts of a smartphone customer, while being unable to access their actual age values, which can be used to infer sensitive information, such as the smartphone customer's

own age. Instead, they should be able to perform statistical analysis on a collection of ages, calculating its count, average, median, mode, and standard deviation. Assume that age is represented as an integer. Now, consider how one can put the aforementioned policy in place by using ObEx. Next, we describe the three main steps of the process: (1) data collection, (2) data transfer, and (3) data analysis.

(1) Data Collection. The data collection step wraps sensitive data in ObEx objects and instructs the ObEx runtime system to transfer the objects to the consumers. Recall that when creating an ObEx object, an access policy must be given. In this case, the producer can define an access policy with the following parameters (two queries of any kind, 10,000 milliseconds lifetime, all query methods permitted with the exception for `sum`). The access policy has to be consistent with the needs of data consumers; otherwise, the provided ObEx objects are of no use to the consumers. Section 7.2 provides details about the ObEx APIs and runtime design.

(2) Data Transfer. The data transfer process is responsible for moving initialized ObEx objects across the network to the consumer sites. To that end, the ObEx runtime encrypts and serializes the sensitive data to a binary stream. The stream is transferred to the destination site, at which the runtime unserializes, decrypts, and reconstructs the stream into an ObEx object instance. The lifetime timer starts immediately after an ObEx object is reconstructed. Section 7.2 describes the ObEx distributed runtime.

(3) Data Analysis. The data analysis process is concerned with executing queries against the sensitive data protected by ObEx objects. In this use case, the telecoms provides a collection of ObEx objects containing the ages of a given customer's contacts. Recall that the access policy allows data consumers to invoke any statistical method with the exception of `sum`, with only two queries permitted. A mobile app can invoke the `average` and `stdDeviation` methods to obtain the necessary information required to be able to make a reasonable

guess whether the customer is likely a migrant. Notice, that ObEx provides the necessary information without revealing the actual ages of the customer's contacts. Furthermore, the sensitive information becomes inaccessible after 10 seconds, thus further preventing potential abuse by nefarious parties. Section 7.2 details the ObEx APIs, while Section 7.4 discusses the performance characteristics of our reference implementation.

To sum up, the raison d'être behind ObEx is to resolve the inherent conflict between the user's privacy needs and the mobile developer's wishes when it comes to managing sensitive data. In this use case, the ObEx protection ensures that the actual ages of phone customers are never revealed to non-trusted parties, which in this case are services supporting mobile applications. These services can still obtain valuable business intelligence by querying ObEx objects, without directly accessing the sensitive data. The following sections discuss the technical details of the ObEx design, implementation, and evaluation.

7.2 Design and Implementation

In this section, we first introduce key design ideas behind ObEx objects, then we present the technical details of our implementation.

7.2.1 Design Overview

In the typical usage scenario described above, the inherent conflict between data producers and consumers is that the former require that their sensitive data be kept private, while the latter wish to leverage some properties of the private data to provide better services and build more intelligent applications. The ObEx mechanism reconciles these conflicting requirements. It keeps the sensitive data invisible, but enables non-trusted clients to query

the data in a controlled way. Once the number of queries or the time limit is exceeded, the ObEx runtime clears the sensitive data, making it inaccessible for any future operations.

(1) Invisible Data. The ObEx architecture keeps its sensitive data in the native layer (managed by the ObEx runtime), never passing it to the managed code (i.e., bytecode) layer. Hence, the sensitive data cannot be copied to regular Java objects. The ObEx API provides methods for serializing and cloning ObEx objects. However, these methods' implementation keeps the sensitive data secure in the native layer. Application clients can invoke any of the predefined ObEx `native` query methods allowed by the policy in place. Thus, the data remains invisible and inaccessible to non-trusted consumers, which derive useful insights about the data's properties to enrich the functionality and user experience of applications and services.

(2) Configurable Expiration Policy. How the sensitive data can be accessed is dictated by a configurable expiration policy that describes three interconnected limitations: max number of accesses, time-to-expiration, and available query methods. The first two limitations determine when the object and its sensitive data should be cleared. The last limitation confirms which query methods non-trusted clients are allowed to invoke.

In addition to query methods, the ObEx API includes meta-query methods, through which the developer can discover what type of data encapsulates a given ObEx object and which query methods can be invoked on the object. In essence, the meta-query methods return information about the access policy, whose restrictions have to be expressed by the producer.

(3) Enforceable Lifecycle Figure 7.2 shows the enforceable lifecycle of ObEx objects. In phase 1, although the sensitive data is hidden in the native layer, non-trusted clients can query the data from the managed layer. In phase 2, the object has cleared itself in the native layer as driven by a given policy, nullifying the data and disabling all future queries.

In phase 3, the object has been completely garbage collected both in the native layer and the managed layer. Currently, our approach supports the lifecycle enforcement in phases 1 and 2. However, in phase 3, the garbage collector only collects the managed portion of ObEx objects; the native layer's sensitive data is cleared based only on the access policy in place. Most likely the ObEx runtime would clear the sensitive data much earlier than the managed code's portion is garbage collected.

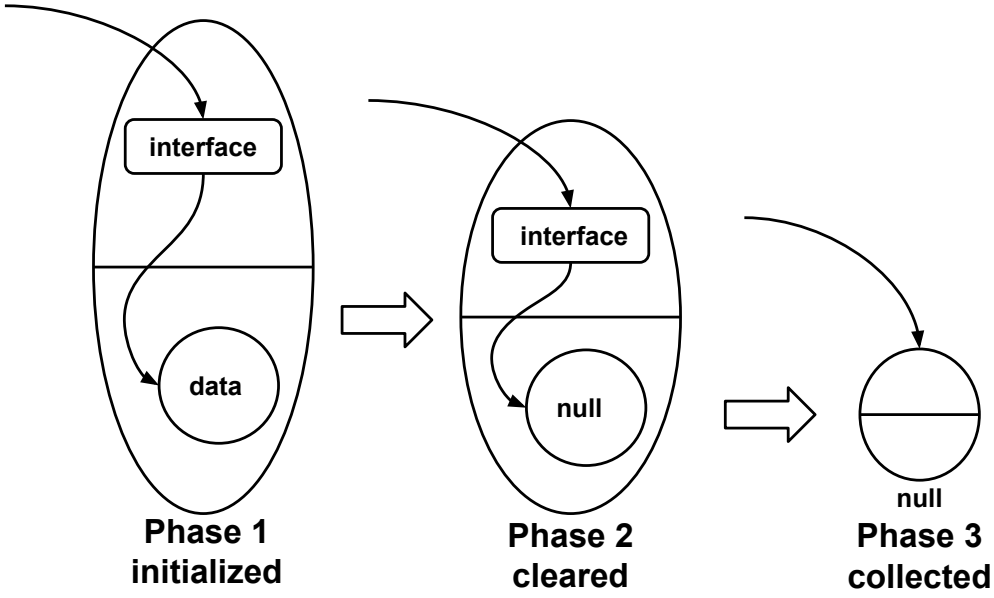

Figure 7.2: Life Cycle of ObEx.

7.2.2 System Architecture

The ObEx system architecture comprises three interacting components, as shown in Figure 7.3: Programming Interface, Local Runtime, and Distributed Runtime. The Programming Interface provides controlled programmatic access to ObEx objects. The Local Runtime is a per-host native library responsible for safekeeping the sensitive data, its lifecycle and operations. The Distributed Runtime is responsible for transmitting ObEx objects across the network securely, while also maintaining their specified access policies.

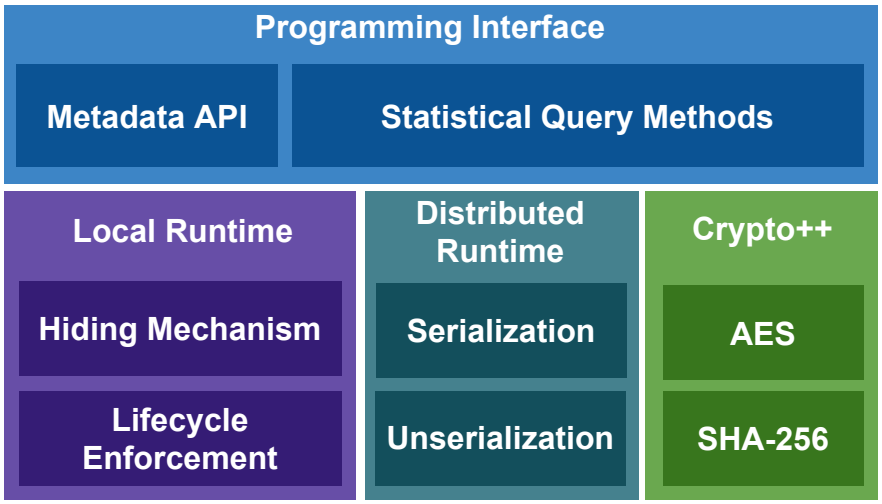

Figure 7.3: Structure of ObEx.

Programming Interface. Both the sensitive data's producer and consumers interact with instances of ObEx objects via their programming interface. The producer passes an instance of sensitive data to the constructor of class ObEx alongside with its access policy. Depending on the policy, consumers can invoke different subsets of the API. These subsets expose various query methods that include statistical and comparison operations. The sensitive data remains invisible and inaccessible to consumers.

API Specifics: Figure 7.4 depicts the main structure of the ObEx class, which reifies the access semantics specified by declarative policies. ObEx provides methods for initializing, querying, deep-copying, and statistical calculations. To control the deep-copying and serialization behavior of ObEx objects, the class is declared as `Externalizable`, taking full control of marshaling its instances.[2] The class includes the `private` field `mID`, operation methods, and `native` method declarations. `mID` uniquely identifies ObEx objects, serving as the key that the API methods use when querying sensitive data. To ensure the `mID`'s uniqueness, its value is the sensitive data's digital signature. Conversely, the query methods are simple

[2]Notice that custom serialization would not reveal the protected sensitive data, as it is managed in the native layer, which is inaccessible to serialization libraries.

wrappers around the `native` functions that interact with the local runtime. Specifically, each `private native` function has a corresponding `public` wrapper method.

We designed other classes to support the functionality provided by ObEx. The abstract class `AccessPolicy` serves as the base class for access policies. Figure 7.4) shows how the API facilitates the implementation of policies via a subclass `CustomPolicy`. The `Methods enum` defines the query methods consumers are permitted to invoke on the invisible sensitive data. The data producer is also required to specify the stored sensitive data's type, which are currently confined to built-in Java primitive types and `String`. This provision is necessary to be able to execute all operations on sensitive data within the native layer, without having to pass the sensitive data to the managed layer.

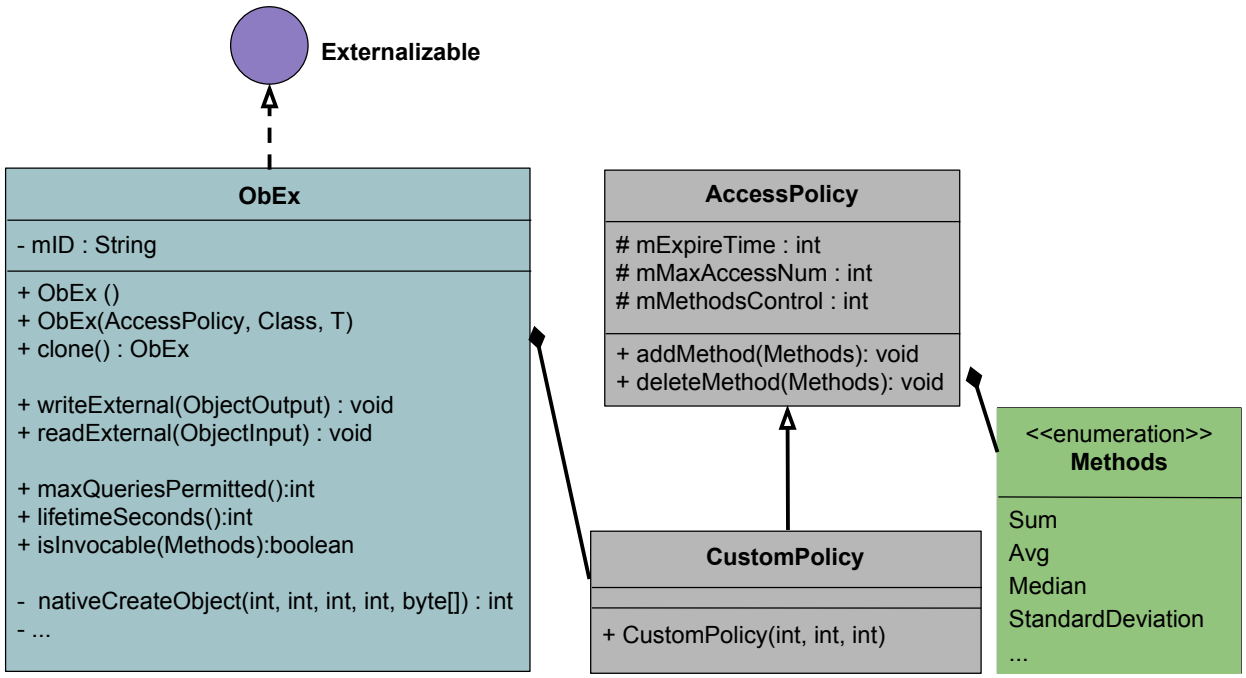

Figure 7.4: Class Diagram of ObEx.

Metadata API: The metadata API appears in Figure 7.5. The metadata provides information about the protection mechanisms of a given ObEx object. The method names are self-documenting. The `getDescription` return a textual representation of the sensitive data

represented by a given ObEx. In the presence of multiple sensitive data feeds, this method can be invoked to differentiate between them. `maxQueriesPermitted` returns an ObEx object's maximum number of queries, while `lifetimeSeconds` returns its time-to-expiration. The `isInvocable` method can be used to determine which query methods are permitted.

```
1 public native String getDescription()
2 public native int maxQueriesPermitted()
3 public native int lifetimeSeconds()
4 public native boolean isInvocable(Methods enm)
```

Figure 7.5: Query Interface for Metadata.

Statistical Query Methods: ObEx supports two types of statistical queries [3]. The first one applies to a single ObEx object, while the second applies to a collection of ObEx objects. Figure 7.6 shows the query methods that can be invoked on a single ObEx instance. The `equals` and `greater` methods can be invoked to compare some consumer-provided data with the invisible data. For example, an ObEx object can contain a secret verification code. The consumer can invoke a limited number of `equals` method to confirm a user. If the number of invocations or the specified lifetime is exceeded, the actual verification code, the sensitive data in this case, will become inaccessible. Consumers can also learn mathematical boundaries of the hidden data by invoking the `greater` method.

```
1 public native boolean <T> equal(T data)
2 public native boolean <T> greater(T data)
```

Figure 7.6: Query Interface for a Single ObEx Object.

For statistical calculations over multiple objects, ObEx provides a set of statistical methods. These methods are all implemented in the native layer. Our design assumes that all the ObEx objects transferred to the same consumer are homogeneous, containing the same type

[3]To prevent information leakage, developers are confined to a fixed set of predefined queries.

of data. The data-intensive application in place determines what data it is (e.g., temperature readings, GPS locations, velocity, etc.) These generic methods return the statistical calculations as determined by the sensitive data on which they operate.

```
1 public static native <T> T sum()
2 public static native <T> T average()
3 public static native int count()
4 public static native <T> T median()
5 public static native <T> T[] mode()
6 public static native <T> T stdDeviation()
7 ...
```

Figure 7.7: Query Interface for Multiple ObEx Objects.

Figure 7.7 shows the query methods that can be invoked on collections of ObEx objects. These methods provide basic statistical functions, including sum, average, count, median, mode, and standard deviation. Some of the functions come in multiple flavors. For example, consumers can both calculate the average of a collection of invisible data, and also can do so with a given value range.

An important property of these queries is that they only include unexpired ObEx objects. The expired objects are dynamically removed from the examined collection. Hence, running the same query in sequence may produce different results. This property can be leveraged to monitor the currently provided and still available sensitive data, protected by ObEx objects.

Local Runtime. The ObEx local runtime hides the sensitive data, enforces the data's lifecycle, and executes query operations. Internally, the local runtime organizes the sensitive data by means of two hash tables, which map ids to data and metadata (Figure 7.8). Data consumers interact with ObEx objects by means of their operations, which are implemented at the native layer. The operations include initialization, cloning, metadata querying, and statistic calculations. Meanwhile, the execution of these operations is governed by the ObEx object's access policy; this policy enforces the object's lifecycle. To support all cryptographic

functionality in ObEx, our implementation integrates a third-party C++ library, Crypto++ [4]. For example, this library provides the SHA-256 [145] algorithm, used to compute a digital signature that serves as a unique id for ObEx objects. To sum up, it is the local runtime that keeps the sensitive data invisible, while enabling consumers to query the data, thus preserving user privacy.

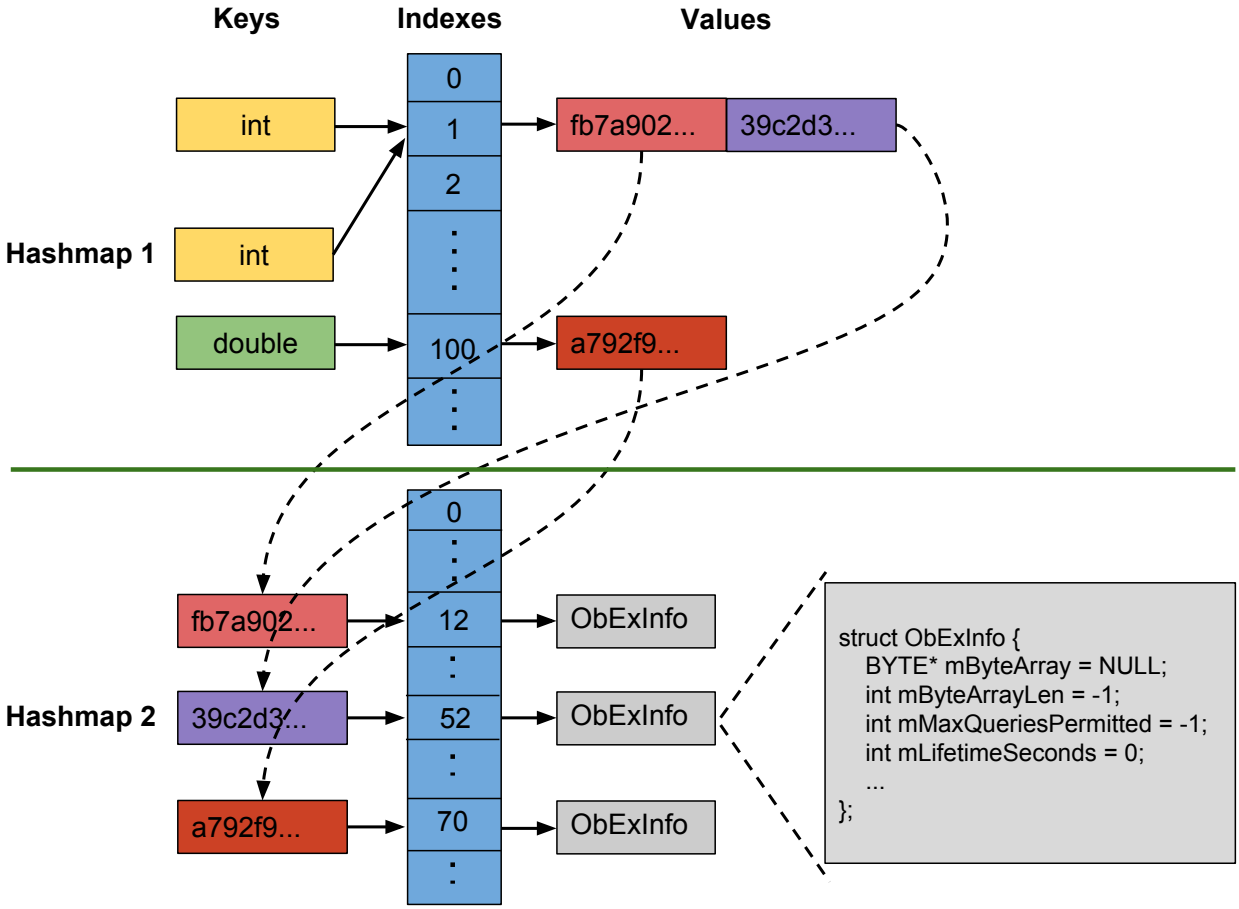

Figure 7.8: Data Structure of ObEx (Hash Tables).

Privacy Preservation Mechanism: Figure 7.9 shows how ObEx preserves user privacy by hiding sensitive data, while enabling consumers to leverage the data's properties. Data producers hand over the sensitive data that needs privacy protection by creating an instance of an ObEx object. The sensitive data is passed as a parameter to the ObEx constructor,

[4]Crypto++® Library 5.6.5 https://www.cryptopp.com/wiki/Main_Page

with the ObEx runtime converting the data to a binary buffer. Consumers can then request a data feed of ObEx objects from the producer.

In response to receiving such a request, the producer initiates a distributed communication, through which the ObEx runtime securely transfers the sensitive data to the requesting consumer, passing the sensitive data to its respective ObEx local runtime. The local runtime maintains the sensitive data at the system level, with all queries performed by means of `native` JNI methods. The ObEx Java methods are simply wrappers over these `native` methods. To sum up, this process ensures that the sensitive data guarded by ObEx remains invisible at the bytecode level. In other words, the producer's sensitive data cannot be accessed directly by the consumers, thus preserving user privacy.

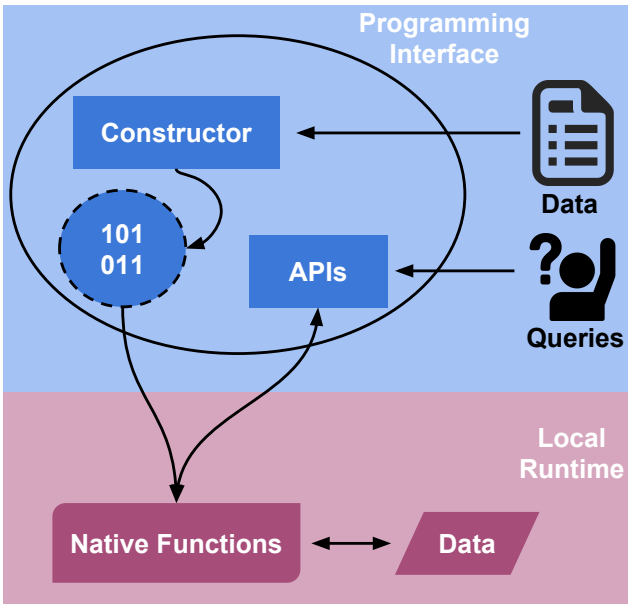

Figure 7.9: Privacy Preservation Mechanism.

Lifecycle Enforcement: In addition to hiding sensitive data, the ObEx runtime enforces the data's lifecycle. That is, the sensitive data remains queryable only for a specified time period; after this period has been exceeded, the data becomes inaccessible for any future queries. To support this lifecycle enforcement policy, the ObEx runtime implements two different

schemes, depending on whether a given ObEx object guards a single sensitive data item or a collection of them. These schemes are realized as follows.

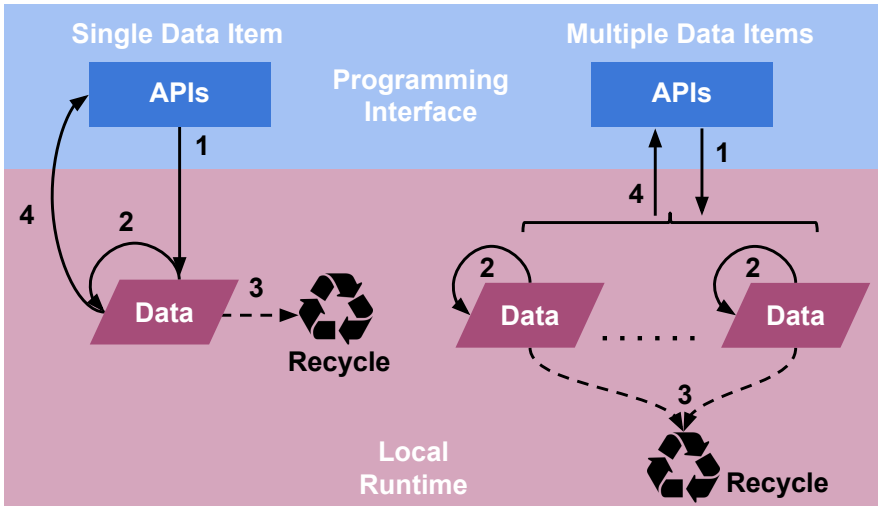

Figure 7.10: ObEx Lifecycle Enforcement Process.

(1) Figure 7.10 shows the lifecycle enforcement process for a single sensitive data item. First, a data consumer initiates an operation on an ObEx object (step 1). For example, the consumer wants to check whether the data `equals` some value. The runtime then checks the object's access policy to determine if (a) the object's lifetime has not passed, (b) the operation is permitted, and (c) the max number of operations has not been exceeded (step 2). If any of these three conditions is unmet, the runtime nullifies the sensitive data (step 3). Finally, the runtime returns the operation's result or raises an exception (step 4).

(2) Figure 7.10 also shows the lifecycle enforcement process for multiple sensitive data items. Different from the single item case, after receiving an invocation (e.g., `average`) (step 1), the runtime checks all the ObEx objects in the collection for the conditions (a) (b) (c) above (step 2). Then, the objects for which these conditions are unmet have their sensitive data nullified (step 3). Later, the operation's result takes into account only the sensitive data of unexpired objects; if no objects remain accessible, a runtime exception is raised (step 4).

Distributed Runtime. The ObEx distributed runtime is responsible for transmitting ObEx objects across the network, while also maintaining their specified access policies. Figure 7.11 shows the entire data transfer procedure. For the data consumers' perspective, the behavior of ObEx object is similar to a regular Java object. The process has the following steps: (1) serialize the object to a binary stream, (2) transfer it to the non-trusted parties via standard web services over the network, (3) unserialize the binary stream and reconstruct the object. However, the ObEx distributed runtime is responsible for serializing / unserializing the objects.

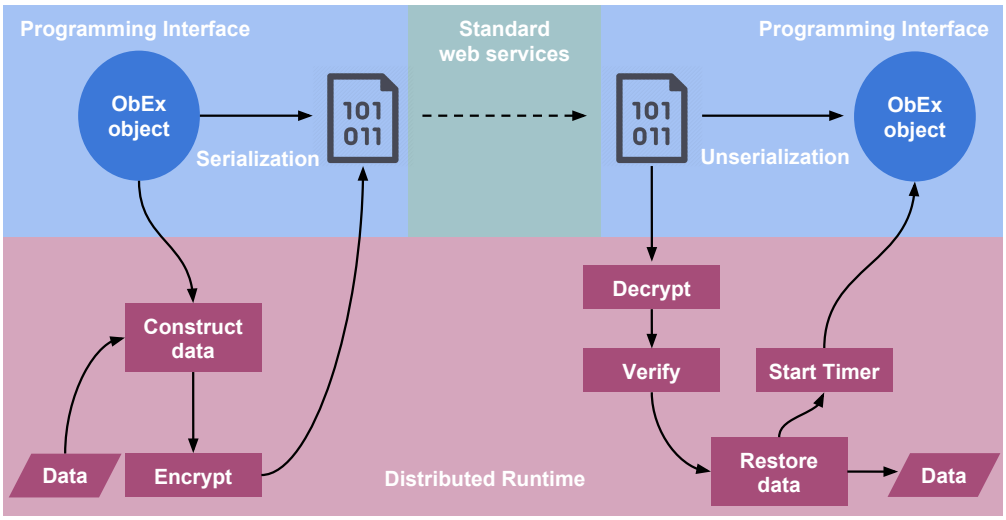

Figure 7.11: Data Transfer Procedure.

(1) During the serialization, the runtime converts the sensitive data and the metadata into a binary stream. After encrypting it via AES algorithm, it returns the result, which can be integrated with the rest of the serialization process in the managed layer.

(2) During the unserialization, the runtime decrypts the arrived binary stream. Then, it recalculates the sensitive data's digital signature, comparing it with the original object's id, so as to verify the integrity and correctness of the ObEx object. Next, hash tables are set up for efficient lookups. Finally, the runtime restarts the expiration timer, thus making the

reconstructed object available for queries.

To sum up, the distributed runtime takes charge of the construction/reconstruction, encryption/decryption, and verification for ObEx objects. These objects can be transferred alongside regular Java objects, while securing their sensitive data and corresponding access policy.

7.2.3 Object-Level Privacy Threats and Defense

OOP prescribes that objects should encapsulate both data and behavior. Hence, object-level privacy threats try to maliciously gain access to the data by invoking private methods or deep copying the entire object. Attackers can use Java reflection API to perpetrate the first two operations, and can deep-copy objects by serializing them to a memory buffer and reading them back.

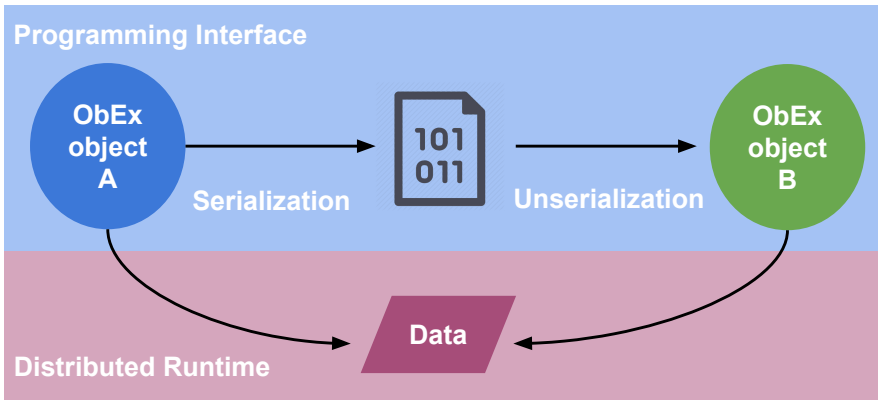

Figure 7.12: Process of Deep-Copying ObEx Object.

ObEx objects are not vulnerable to the reflective access vulnerability, as they keep no sensitive data at the bytecode level. Deep-copying, on the other hand, can create copies that can be forwarded to multiple sites, multiplying the specified access privileges. In this experiment, we emulate this copying attack to check if the ObEx design can defend against it.

Figure 7.12 show how when deep-copying an ObEx object, only the managed layer's content is duplicated, while the sensitive data in the native layer is not. In other words, the copies alias the same sensitive data. The presence of aliases has no effect on the sensitive data's lifecycle.

7.3 Discussion

To develop an effective application that ensures sensitive data privacy, the data producer must determine (1) how long the sensitive data should remain queryable? (2) how many queries should it permit? (3) which queries should be allowed? Obviously the answers to these questions are application-specific. We discuss each question, demonstrating the issues involved with a concrete example.

7.3.1 Time-to-Expiration

To define a reasonable lifetime value for sensitive data, a data producer must consider: (1) whether the sensitive data should be observed in real-time, and (2) for how long the access to the data should be granted to consumers.

As an example demonstrating the first question, consider intelligent navigation applications, which require in-time GPS data to compute current traffic information. That is, only the latest data is valuable, while the outdated data should be excluded from the statistical calculations. Meanwhile, the frequency of real-time data arrival can determine the lifetime as well. For example, if a health tracker provides heartbeats every 30 seconds, the lifetime of $> 90{,}000$ milliseconds would enable calculating the averages of 3 consecutive heartbeats.

As an example demonstrating the second question, consider sending a hidden verification

code, which should expire in 30 seconds; or limiting access to the license of commercial software to one hour; or leasing a marketing dataset for 24 hours. Hence, the actual business properties of sensitive data determine the lifetime value that data producers should assign to the data's ObEx object.

7.3.2 Max Number of Accesses

Statistical privacy provides mathematical formulæ to help determine the max number of accesses that preserves user privacy [97]. However, the number of accesses can be determined via business analysis in many cases. We further explicate this issue in section 7.1.

Statistical Model. Recall the motivating use case, in which a person communicating with contacts of similar age is likely to be a local. By knowing the average and standard deviation of a customer's contacts, enterprises can infer whether the customer is likely a local or migrant. If a customer is already determined to be a local, this fact can be leveraged to also learn the customer's age by continuously invoking methods `greater` or `equal`. Next, we describe two different algorithmic approaches that a malicious data consumer can exploit to guess a customer's age, a data item whose privacy we want to preserve.

Binary Search: Binary search can find an integer between $0...n$ in $\log_2 n$ comparisons. Assume that a customer's age can range between 1 and 100, while the age's probability distribution is unknown in advance. By using binary search, one can guess the age's value in $\log_2 100 \approx 6.6$ queries. Thus, to reduce the risk of leaking the age information to non-trusted parties, the producer should set the maximum number of permitted `greater` queries to fewer than 6.

Guessing Entropy: In probability theory, the guessing process is an "uncertain event" to developers, and the Shannon entropy [142] can be used to determine the uncertainty of

information. Christian Cachin [39] proposed a guessing entropy for estimating the expected number of guesses under an optimal guessing strategy. In this work, we apply this guessing entropy to determine the number of accesses data providers should specify in the access policy of an ObEx object.

Let χ be a probability distribution, and x be a specific event that can be considered as the correct age in our case. Then, let X be the random variables representing the age guessed by data consumers. Meanwhile, X has N possible values, and the probability of X is P_X, so that the elements of P_X will be $p_1, p_2, ..., p_N$.

Therefore, our case could be described based on the mathematical notations above: the age (x) has been already stored in the ObEx object. The developer has collected N possible age values (X_i) with probabilities $p_1, p_2, ..., p_i, ..., p_N$. Thus, "is X_i equal to x?" will be the `equal` querying process of data consumers. In addition, we assume P_X is a monotonically decreasing sequence, which is $p_1 \geq p_2 \geq ... \geq p_{N-1} \geq p_N$. Then, the optimal guessing strategy is to guess the most likely value first, following the sequence of P_X. The guessing entropy [39] is:

$$E[G(X)] = \sum_{i=1}^{N} p_i \cdot i \tag{7.1}$$

The formula calculates the expected number of accesses the developer needs to query an ObEx object to obtain the correct age under the optimal guessing process.

An Example. An ObEx object represents the hidden age value as an integer, whose range is 0...100. Assume the local's age is 25, while a consumer has collected a data set including the possible age values of $30, 25, 28, 26$ with the respective probabilities of $40\%, 30\%, 20\%, 10\%$. Based on the guessing entropy, the expected number of `equal` quires is $40\% \cdot 1 + 30\% \cdot 2 + 20\% \cdot 3 + 10\% \cdot 4 = 2$. Therefore, the access policy should set the number of accesses to less

than 2 to reduce the risk of leaking the sensitive age data to non-trusted parties.

7.3.3 Permitted Query Methods

To systematically determine which query methods should be permitted on a given ObEx object, data producers should follow the following procedure:

(1) Inspect Statistical Significance: The semantics of sensitive data determines which operations can be meaningfully applied to it. For example, it would be absurd to allow clients to only compute the `sum` of observed individual body temperatures or to compute the `average` of GPS locations.

(2) Consider Privacy Requirements: Data producers should limit query methods to meet the requirement of data privacy. In our use case, allowing consumers to `sum` a collection of ages is harmless, but for financial data, this query can reveal sensitive information.

(3) Exclude Composite Methods: Notice that some statistical calculations can be substituted by a combination of several other operations. For example, one can calculate the `average` of a collection by combining `sum` and `count`. In this case, if the former method is inaccessible, then the latter ones should be excluded as well.

7.4 Evaluation

We first evaluate various performance characteristics of the reference implementation of ObEx; we then show how ObEx defends against a deep-copy attack, intended to subvert the

ObEx mechanism for protecting sensitive data; finally, we report on the lessons learned from the evaluation.

To assess the performance characteristics of ObEx, we first measure the total time it takes to create, serialize, and statistical query ObEx objects, which encapsulate sensitive data of various sizes. We then measure the total memory consumption at runtime to determine the actual limit on the number of ObEx objects that can be created without exhausting the total JVM memory.

7.4.1 Control Group Choice and Runtime Environment

Table 7.1: Runtime Environment.

OS	ubuntu 16.04 LTS, 64-bit
Memory	11.5 GiB
Processor	Intel Core i5-3210M CPU @ 2.50GHz * 4
JDK	OpenJDK 1.8.0_131
JVM	OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)

The ObEx object can be considered efficient if its performance is competitive with other existing APIs provided by Java platform. Thus, we compare the speed of initializing and serializing ObEx objects against three Java APIs, which provide relevant functionalities, including `SealedObject`, `SignedObject`, and `String` [5]. We include `SealedObject` and `SignedObject` as other internal protection mechanisms that improve data privacy. We include `String`, as it is the most commonly used object type with similar features (storing a byte stream). To ensure a fair comparison, we configure the `SealedObject` and `SignedObject` objects with the same encryption algorithm (i.e., `AES` and `SHA-256`) used by ObEx. Table 7.1 shows the runtime environment used for all experiments.

[5] Java Platform, Standard Edition 7 API Specification
https://docs.oracle.com/javase/7/docs/api/

7.4.2 Performance

We compare the respective performance of our subjects in terms of the total runtime and memory consumption. The time consumed by initialization, serialization / unserialization and statistical queries is measured by means of `System.currentTimeMillis`. A Java instrumentation package, running an agent monitoring the memory status at runtime, is used to measure memory consumption.

To determine how the size of sensitive data affects performance, we evaluate a data series of different sizes, ranging from 10 bytes to 1 megabytes. Furthermore, to increase the evaluation's reliability, we repeat each measurement for 1,000 times and average the results.

(1) Time consumption.

Instantiation: Instantiation comprises memory allocation, default value initialization, shared library setup, etc. ObEx invokes a native function to initialize ObEx objects, which allocates variables and data structures, generates the metadata (e.g., current time), calculates the unique object ID, passing it to a field in the managed layer. To measure the time consumed, we record the time before and after creating the object, with Figure 7.13 showing the results.

As expected, the time consumed increases as sensitive data grows in size. In contrast to the control group, which experiences a sharp spike in execution time as the data reaches 1 megabyte in size, ObEx maintains the average execution time of (11.913ms). Although ObEx takes longer to initialize for small objects than `String` and `SealedObject`, it maintains stable performance characteristics irrespective of the sensitive data's size.

Serialization / Unserialization: ObEx objects are serialized/unserialized when transferred across the network. Normally, an object can be serialized to a binary stream and sent to another device or server. When it arrives to the destination, it is unserialized and recon-

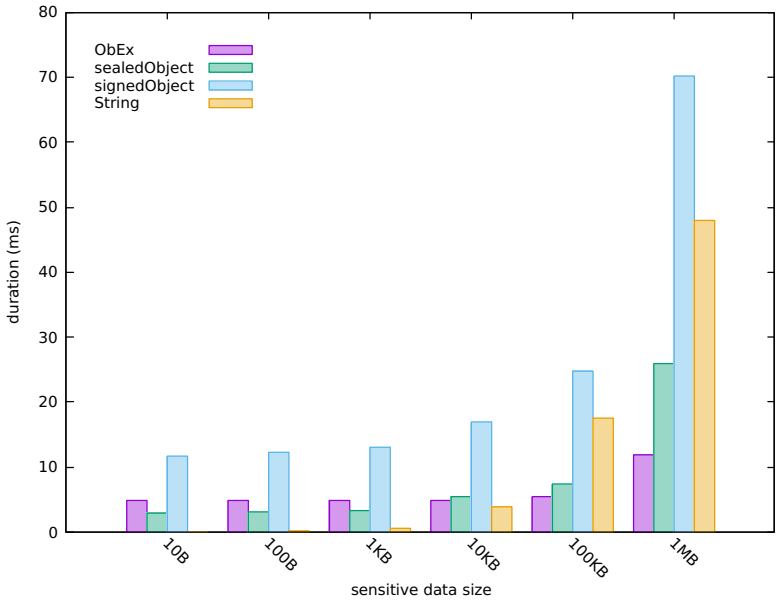

Figure 7.13: Performance of Instantiation.

structed to an isomorphic object, with the same data fields. When transferring ObEx objects across the network, the runtime first gathers the relevant sensitive data and metadata, and then encrypts and combines them into a byte array as part of serialization. Unserialization reverses the process.

Figure 7.14 shows the total time consumed by the serialization process. Due to the time taken by assembling and encrypting data, the objects in the control group outperform the ObEx objects. However, for `String` objects, as the contained data increases in size, the time consumed grows rapidly. When the data size reaches 1 megabyte, ObEx (31.32ms) surpasses `String` (49.341ms).

Meanwhile, Figure 7.15 depicts the time consumed by the unserialization process. ObEx outperform both `SealedObject` and `SignedObject` for small data sizes (from 1B to 10KB). For 1MB, ObEx (16.239ms) only outperforms `String` (26.23ms).

As part of their serialization/unserialization ObEx objects go through expensive operations,

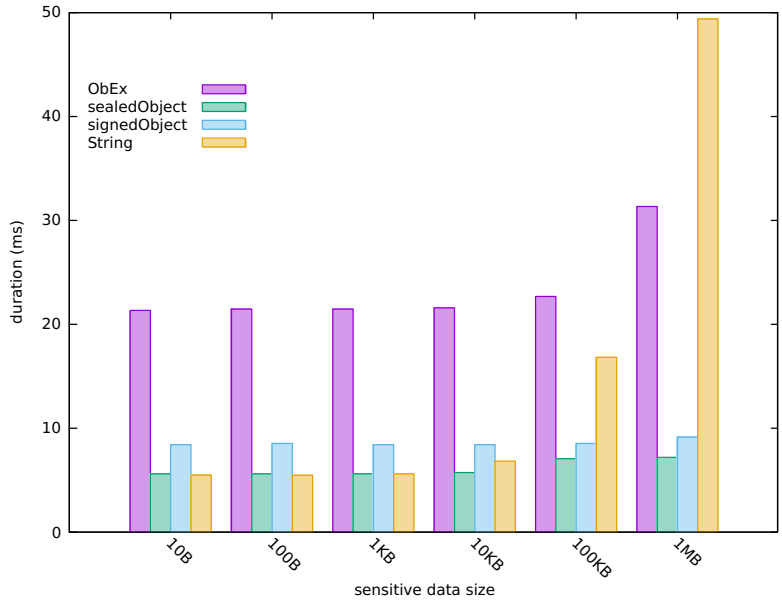

Figure 7.14: Performance of Serialization.

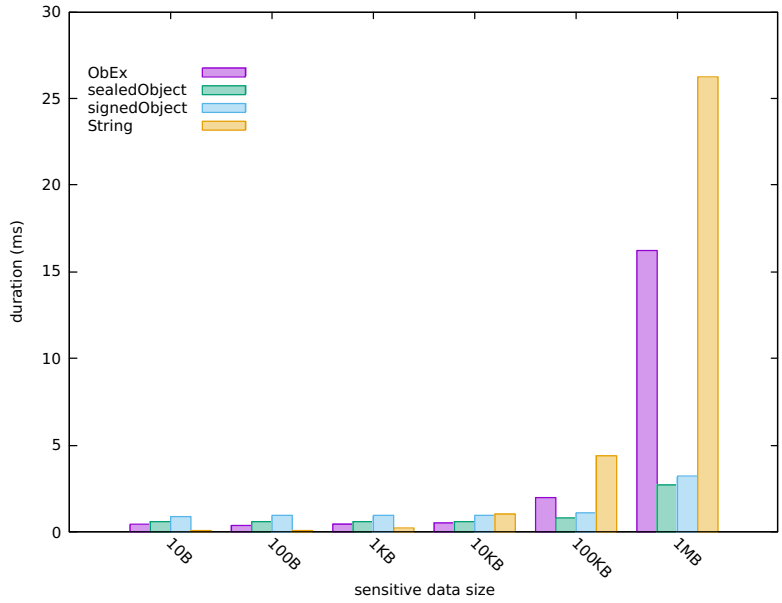

Figure 7.15: Performance of Unserialization.

such as encryption/decryption, construction/reconstruction, digital signature generation, and verification, thus losing out to the control group, for which these processes are not as involved. ObEx still outperforms `String`. Thus, one can conclude that ObEx shows

satisfying performance characteristics as compared to the control group.

(2) Statistical Queries.

Here we measure the total time taken by the statistical calculations a data consumer may want to perform on the ages of a phone user's contacts, including average, count, median, mode, and standard deviation. We first randomly generate age numbers to create a collection of ObEx objects. We then record the time taken to perform each statistical method. Table 7.2 shows that the time increases linearly with the growth of the number of ObEx objects, when none of the object is nullified. Notice that the ObEx runtime first checks if an object may have expired before including its data into a given statistical calculation, thus incurring additional processing time. As future work, we plan to investigate how to parallelize the processing of large collections.

Table 7.2: Time Consumption of Statistical Queries.

Num	Avg	Count	Median	Mode	Std.
10	≈ 0ms	≈ 0ms	≈ 0ms	1ms	≈ 0ms
100	4ms	5ms	4ms	5ms	4ms
1000	543ms	551ms	553ms	554ms	546ms

7.4.3 Memory Consumption

For data-intensive applications, it is important to ensure that ObEx is memory efficient. Therefore, we first compare the memory consumed by ObEx and the control group, containing data items ranging between 10 bytes and 1 megabytes in size. We investigate the upper limit on the number of ObEx objects that can be created in a single, default configuration JVM.

(1) Runtime allocation.

To measure memory consumption, we execute an agent JAR file containing an implementation of `premain-class`. By invoking `java.lang.instrumentget.getObjectSize`[6], we can accurately estimate the runtime memory allocated for each object. However, this method can only measure the amount of memory consumed by the specified object. In other words, the fields inherited from superclasses or the actual size of instances in a reference array will be omitted. To address this problem, we sum all object fields by recursively traversing them through Java reflection.

Table 7.3: Memory Allocation of ObEx.

Data Size	native Layer	Managed Layer
10 B	94 B	96 B
100 B	184 B	96 B
1 KB	1108 B	96 B
10 KB	10324 B	96 B
100 KB	102484 B	96 B
1 MB	1048660 B	96 B

Since ObEx objects run in both the managed and native layers, we measure the memory consumed in both of them. In the managed (i.e., bytecode) layer, the memory allocated for ObEx is fixed to 96 bytes in all different cases of sensitive data sizes (Table 7.3). The reason is that only the `mID` and several interfaces of wrapper method are stored in the managed layer. In the native layer, ObEx objects consume the amount of memory proportional to the size of the sensitive data they contain. Besides, ObEx objects have metadata attached to them, containing their lifecycle policy, including creation time, the number of accesses, and data type. We sum the memory consumed by all these data items and compare the result with the control group (Table 7.4).

Table 7.3 shows that the memory consumed by the ObEx objects is almost the same as that

[6]Package java.lang.instrument
http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html

of `SealedObject` and `SignedObject`, which are proportional to the size of the sensitive data they encapsulate. `String` objects allocate two times as much memory as the other subjects for the 1MB data size. These results indicate that ObEx never consumes excessive volumes of memory. More importantly, the ObEx runtime manages the memory allocated for ObEx objects explicitly rather than relying on garbage collection as the control group does.

Table 7.4: Memory Allocation of ObEx and Control Group.

Size	ObEx	SealedObject	SignedObject	String
10 B	190 B	120 B	272 B	40 B
100 B	280 B	200 B	360 B	216 B
1 KB	1204 B	1128 B	1288 B	1960 B
10 KB	10420 B	10344 B	10504 B	19616 B
100 KB	102580 B	102504 B	102664 B	196664 B
1 MB	1048756 B	1048680 B	1048840 B	2014080 B

(2) Memory Limitation of ObEx Objects.

As compared with Java objects, ObEx objects keep the majority of data in the native layer. In other words, if someone continuously creates an ObEx object with infinite expiration time, the runtime memory will be exhausted. In general, this limitation highly depends on the size of physical memory. We can observe this phenomenon by creating an ObEx object with 1 megabytes sensitive data in an infinite loop, setting the expiration policy to infinity. Table 7.5 shows the percentage of runtime memory consumed by continuously creating ObEx objects. After the number increased near to 10,000, the test environment hangs without responding. This result shows that the maximum possible number of ObEx objects depends on the size of the available physical memory. Our runtime environment, with 11.5 GiB physical memory, can support up to 10,000 objects, each containing 1 megabyte of sensitive data.

Table 7.5: Memory Limitation of ObEx Objects.

Data Size	Number of Objects	%MEM
1 MB	2000	17.6
1 MB	4000	34.7
1 MB	6000	51.7
1 MB	8000	68.0
1 MB	≈10000	N/A

7.4.4 Lessons Learned

Despite the privacy-preservation benefits of ObEx demonstrated above, to fully realize its potential, we argue that it should be supported natively by the JVM. This native support would help (1) complicate privilege escalation and library substitution attacks, (2) integrate the ObEx life cycle management with garbage collection, and (3) avoid the dependence on third-party security libraries. We next explain these expected benefits.

Complicating Privilege Escalation and Library Substitution Attacks: Because the ObEx local runtime is a shared library loaded by the JVM, privilege escalation (e.g., `root` account in Linux) can directly access the sensitive data in memory. In addition, an attack can replace the ObEx runtime with a malicious version at load time.

Integrating with the JVM can increase the integrity of the ObEx privacy-protection mechanism, as the JVM security mechanism makes it a hard target for attackers[7] [8]. Although the highest privilege attacks still get unrestricted access to the system, the success rate of such attacks is rare for properly administered systems.

Integrating with JVM Garbage Collection: After the ObEx runtime clears sensitive data, the memory allocated for its ObEx object should become reclaimable immediately.

[7] JVM Specification http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[8] Secure Computing with Java: Now and the Future
http://www.oracle.com/technetwork/java/javaone97-whitepaper-142531.html

However, to be able to mark ObEx objects as available for garbage collection requires JVM integration.

Avoiding Dependencies on Third-Party Security Libraries: Despite the existence of the JDK package `javax.crypto.*` [9], the ObEx runtime relies on a third-party C++ library (Crypto++), while integrating with the JVM would allow using its built-in crypto libraries. To facilitate the proposed integration, we have started incorporating ObEx into the OpenJDK HotSpot VM. We have already added the programming interface. We are still working on integrating our native library. Appendix provides details of this integration effort.

7.5 Conclusions

The advent of mobile, wearable, and IoT devices has generated a deluge of data, much of which has some privacy restrictions. There is an inherent conflict between the end users contributing this data and commercial enterprises. The end users want to keep their sensitive data private, while the enterprise would like to use this data to provide intelligent, context-sensitive services and applications.

In this work, we argue that innovations in programming technology can help reconcile these two conflicting agendas: sensitive data can be kept private, while enterprises can still derive valuable business intelligence from the data. We show how we designed, implemented, and evaluated ObEx, a novel programming abstraction that keeps sensitive user data invisible, while controlling its lifecyle and querying policy. The ObEx provides programming interfaces to perform statistical computations, so as to enable developers to build intelligent mobile and IoT applications.

[9] https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

We have discussed realistic scenarios and use cases that apply ObEx to preserve data privacy. We have discussed the ObEx design, implementation, and programming model. Our evaluation also shows that ObEx exhibits satisfying performance characteristics. To maximize the positive impact, we advocate that the support of ObEx objects be incorporated into the Java Virtual Machine and exposed via a standard API. To that end, we have motivated why ObEx should be integrated with Java Virtual Machine, as well as discussed (in Appendix) how the ObEx programming APIs can be incorporated into OpenJDK HotSpot VM.

Chapter 8

Future Work

As discussed in the chapters above, this dissertation's research innovates in the software engineering space to understand and manage SPI. By applying our approaches, techniques, and tools, a developer can identify SPI functions and variables, and also protect them from data leakage. Nevertheless, it would be unrealistic to expect that this dissertation has addressed all problems pertaining to protecting SPI. For example, whenever mobile apps exchange data, the process might still incur data leakage risks that arise from the possibility of attacks against the communication channel, including interception, eavesdropping, and permission escalation. Some untrusted communication parties may strive to increase their competitive advantage at other parties' expense via data pilfering. To address these problems, we propose to also innovate in the middleware design space to come up with new designs that can mitigate the data leakage risks while still allowing the mutually untrusted parties to exchange data:

(1) Toward Secure Message-Based Communication: In modern mobile platforms, inter-component data exchange is afflicted by data leakage attacks, through which untrustworthy apps access the transferred data. Existing defenses are overly restrictive, as they block all suspicious data exchanges, thus preventing any app from receiving data. To better secure inter-component data exchange, we aim to design and implement a model that strengthens security, while also allowing untrusted-but-not-malicious apps to execute their business logic. Our on-going model introduces two novel mechanisms: hidden transmission

and polymorphic delivery. Sensitive data are transmitted hidden in an encrypted envelope. Their delivery is polymorphic: as determined by the destination's trustworthiness, it can be delivered no data, raw data, or encrypted data.

(2) Toward Privatize On-Device Data Sharing: To maximize customer stickiness, modern mobile services (e.g., advertisement, navigation, healthcare) must be personalized for individual users. To personalize their services, app providers continuously collect sensor data and share it with others, either locally or via cloud. However, the existing mechanisms for sharing sensor data can be exploited by data leakage attacks, thus compromising the privacy of app providers. To address this problem, we plan to design and implement a framework for *differentially privatized on-device sharing of sensor data*, through which app providers can safely collaborate with each other to personalize their mobile services.

Chapter 9

Summary and Conclusions

The data leakage, exfiltration, and tampering attacks motivate research efforts aimed at creating effective mechanisms for protecting sensitive business logic, algorithms, and data. Although SPI protection has been studied extensively from the security perspective, this dissertation research revisits this problem from a software engineering angle, with the goal of providing novel software technologies that can better support developers. This dissertation research innovates in the software engineering space to understand and manage SPI. Specifically, we (1) design and develop program analysis and programming support for inferring the usage semantics of program constructs, with the goal of helping developers understand and identify SPI (i.e. VarSem); (2) provide powerful programming tools that transform code automatically, with the goal of helping developers effectively isolate SPI from the rest of the codebase (i.e., RT-Trust and TEE-DRUP); (3) provide programming mechanism for distributed managed execution environments that hides SPI, with the goal of enabling components to exchange SPI safely and securely (i.e., ObEx). The overriding goal of this research is to contribute to the software development of programming abstractions, automated program analysis and transformation, thus establishing a secure, understandable, and efficient foundation for protecting SPI.

Bibliography

[1] CVE-2015-8944, 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8944.

[2] CVE-2016-9103, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9103.

[3] CVE-2017-1500, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1500.

[4] CVE-2017-17672, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17672.

[5] CVE-2017-5239, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5239.

[6] CVE-2018-15752, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15752.

[7] CVE-2018-6412, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6412.

[8] CVE-2018-9489, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9489.

[9] Mirror of official LLVM git repository, 2018. https://github.com/llvm-mirror/test-suite.

[10] Common Vulnerabilities and Exposures, 2019. https://cve.mitre.org/.

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

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

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

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

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

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

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

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

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

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

[21] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[22] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[23] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016.

[24] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419, 2018.

[25] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[26] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao 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*, pages 23–34. ACM, 2017.

[27] antlersoft. Browse-by-Query, 2011. http://browsebyquery.sourceforge.net/.

[28] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[29] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, et al.

SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 689–703, 2016.

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

[31] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2018.

[32] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[33] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium*, pages 1213–1227, 2018.

[34] Jorge Blasco, Thomas M. Chen, Igor Muttik, and Markus Roggenbach. Wild android collusions. 2016.

[35] Gregory Bollella and James Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[36] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[37] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza

Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

[38] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, 2008.

[39] Christian Cachin. *Entropy measures and unconditional security in cryptography*. PhD thesis, Swiss Federal Institute of Technology in Zurich, 1997.

[40] Guang Chen, Yuexing Wang, Min Zhou, and Jiaguang Sun. Vfql: combinational static analysis as query language. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 378–381, 2019.

[41] Yue Chen, Mustakimur Khandaker, and Zhi Wang. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 334–345, 2017.

[42] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11):1471–1482, 2012.

[43] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[44] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[45] Clang Front End for LLVM Developers. Clang static analyzer, 2019. https://clang-analyzer.llvm.org/.

[46] Tal Cohen, Joseph Gil, and Itay Maman. Jtl: the java tools language. *ACM SIGPLAN Notices*, 41(10):89–108, 2006.

[47] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[48] CVE site. CVE-2011-4120, 2011. https://cvesite.com/cves/CVE-2011-4120.

[49] CVE site. CVE-2019-12210, 2019. https://cvesite.com/cves/CVE-2019-12210.

[50] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[51] drkblog. findmacs, 2018. https://github.com/drkblog/findmacs.

[52] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[53] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[54] Peter Eisentraut. emailaddr type for PostgreSQL, 2015. https://github.com/petere/pgemailaddr.

[55] Zheran Fang, Weili Han, and Yingjiu Li. Permission based Android security: Issues and countermeasures. *computers & security*, 43:205–218, 2014.

[56] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

[57] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth SOUPS*, page 3. ACM, 2012.

[58] Stephan Flake and Wolfgang Mueller. An OCL extension for real-time constraints. In *Object Modeling with the OCL*, pages 150–171. Springer, 2002.

[59] Ira R Forman and Nate Forman. Java Reflection in action (in Action series). 2004.

[60] Matthew Fredrikson and Benjamin Livshits. Zø: an optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 909–924. USENIX Association, 2014.

[61] Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, Salil Vadhan, and Jack Murtagh. Psi (Ψ): a private data sharing interface. In *Theory and Practice of Differential Privacy*, New York, NY, 2016 2016.

[62] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, pages 299–316, 2009.

[63] Narain Gehani and Krithi Ramamritham. Real-time Concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, 1991.

[64] Edward M Gellenbeck and Curtis R Cook. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*, pages 65–81. Ablex Publishing, Norwood, NJ, 1991.

[65] GlobalPlatform. GlobalPlatform, TEE system architecture, technical report, 2011. https://www.globalplatform.org/specificationsdevice.asp.

[66] GlobalPlatform Device Technology. TEE client API specification, June 2010. https://www.globalplatform.org/specificationsdevice.asp.

[67] GlobalPlatform Device Technology. Trusted user interface API, June 2013. https://www.globalplatform.org/specificationsdevice.asp.

[68] GlobalPlatform Device Technology. TEE internal core API specification, June 2016. https://www.globalplatform.org/specificationsdevice.asp.

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

[70] Li Gong and Gary Ellison. *Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[71] Google. Encrypted BigQuery client, 2018. https://github.com/google/encrypted-bigquery-client.

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

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

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

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

[76] Google Inc. gRPC a high performance, open-source universal RPC framework, 2017. https://grpc.io.

[77] Google Inc. gperftools, 2018. https://github.com/gperftools/gperftools.

[78] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[79] Hunter Gregal. MimiPenguin 2.0, 2019. https://github.com/huntergregal/mimipenguin.

[80] Arjun Guha, Jean-Baptiste Jeannin, Rachit Nigam, Jane Tangen, and Rian Shambaugh. Fission: Secure dynamic code-splitting for JavaScript. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[81] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *International Conference on Generative Programming and Component Engineering*, pages 37–56. Springer, 2003.

[82] John W Holford, William J Caelli, and Anthony W Rhodes. Using self-defending objects to develop security aware applications in java™. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 341–349. Australian Computer Society, Inc., 2004.

[83] Cay S Horstmann. *Scala for the Impatient*. Pearson Education, 2012.

[84] Einar W Høst and Bjarte M Østvold. Debugging method names. In *European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[85] Pao-Ann Hsiung. Real-time constraints. In *Institute of Information Science, Academia Sinica, Taipei*, 2001.

[86] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPOR}: Precise and scalable sensitive user input detection for Android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[87] 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.

[88] Intel. The Edger8r tool, 2018. https://software.intel.com/en-us/sgx-sdk-dev-reference-the-edger8r-tool.

[89] Intel. Enclave definition language file syntax, 2018. https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax.

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

[91] Intel. Intel software guard extensions SDK - string functions, 2018. https://software.intel.com/en-us/sgx-sdk-dev-reference-string-functions.

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

[93] Yutaka Ishikawa and Hideyuki Tokuda. *Object-oriented real-time language design: Constructs for timing constraints*, volume 25. ACM, 1990.

[94] J. Karau. phone number scanner, 2014. https://github.com/wittycoder/phone_number_scanner.

[95] Farnam Jahanian and Ambuj Goyal. A formalism for monitoring real-time constraints at run-time. In *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pages 148–155. IEEE, 1990.

[96] Lin Jiang, Hui Liu, and He Jiang. Machine learning based recommendation of method names: How far are we. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–614. IEEE, 2019.

[97] Daniel Kifer and Bing-Rong Lin. An axiomatic view of statistical privacy and utility. *Journal of Privacy and Confidentiality*, 4(1):2, 2012.

[98] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 839–858. IEEE, 2016.

[99] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–10, 2018.

[100] KYLIN Information Technology Co., Ltd. Biometric Authentication, 2019. https://github.com/ukui/biometric-authentication.

[101] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

[102] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.

[103] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, P Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for Intel SGX. In *USENIX ATC*, 2017.

[104] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 359–376. IEEE, 2015.

[105] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[106] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371. ACM, 2017.

[107] Xing Liu, Jiqiang Liu, Wei Wang, Yongzhong He, and Xiangliang Zhang. Discovering and understanding Android sensor usage behaviors with data flow analysis. *World Wide Web*, 21(1):105–126, 2018.

[108] Yin Liu, Kijin An, and Eli 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*, pages 175–187, 2018.

[109] Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: Automated refactoring for different trusted execution environments under real-time constraints. *Journal of Computer Languages*, 56:100939, 2020.

[110] Yin Liu and Eli Tilevich. Varsem: Declarative expression and automated inference of

variable usage semantics. In *Proceedings of 19th International Conference on Generative Programming: Concepts Experiences (GPCE 2020)*.

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

[112] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[113] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[114] John M. and Isayah R. Intel software guard extensions part 3: Design an application, 2016. https://software.intel.com/en-us/articles/software-guard-extensions-tutorial-series-part-3.

[115] Kenny MacDermid. wdpassport-utils, 2016. https://github.com/KenMacD/wdpassport-utils.

[116] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 365–383, New York, NY, USA, 2005. Association for Computing Machinery.

[117] Microsoft. Always Encrypted, 2019. https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/

always-encrypted-database-engine?redirectedfrom=MSDN&view=sql-server-ver15.

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

[119] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *ACM SIGPLAN Notices*, volume 49, pages 411–423. ACM, 2014.

[120] Michael Mimoso. Mobile app collusion can bypass native android security, 2016. https://threatpost.com/mobile-app-collusion-can-bypass-native-android-security/121124/.

[121] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, 2012.

[122] Subhas C Misra and Virendra C Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International Conference on Computational Science and Its Applications*, pages 724–732. Springer, 2003.

[123] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[124] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 993–1008, 2015.

[125] Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of" personally identifiable information". *Communications of the ACM*, 53(6):24–26, 2010.

[126] NetBeans. Jackpot, 2012. http://wiki.netbeans.org/Jackpot.

[127] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. Combining program analysis and statistical language model for code statement completion. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–721. IEEE, 2019.

[128] OP-TEE. Open portable trusted execution environment, 2018. https://www.op-tee.org/.

[129] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch.*, 2016:591, 2016.

[130] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[131] PX4 Dev Team. PX4, 2018. http://px4.io/.

[132] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, 2019.

[133] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from'big code'. *Communications of the ACM*, 62(3):99–107, 2019.

[134] Anil Kumar Reddy, Periyasamy Paramasivam, and Prakash Babu Vemula. Mobile secure data protection using emmc rpmb partition. In *Computing and Network Communications (CoCoNet), 2015 International Conference on*, pages 946–950. IEEE, 2015.

[135] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–22, 2017.

[136] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of Android applications for trusted execution environments. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 923–934. IEEE, 2016.

[137] Prasanna K Sahoo, SAKC Soltani, and Andrew KC Wong. A survey of thresholding techniques. *Computer vision, graphics, and image processing*, 41(2):233–260, 1988.

[138] Luciano Sampaio and Alessandro Garcia. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, 113:337–361, 2016.

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

[140] Paul M Schwartz and Daniel J Solove. The pii problem: Privacy and a new concept of personally identifiable information. *NYUL rev.*, 86:1814, 2011.

[141] Alexander Senier, Martin Beck, and Thorsten Strufe. Prettycat: Adaptive guarantee-controlled software partitioning of security protocols. *arXiv preprint arXiv:1706.04759*, 2017.

[142] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[143] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 161–174. ACM, 2006.

[144] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[145] Secure Hash Standard. Fips pub 180-2. *National Institute of Standards and Technology*, 2002.

[146] N SUMALATHA and SHAIK REHMATHUNNISA NAGA. A trusted hardware based database with privacy and data confidentiality. 2018.

[147] Wesley Tansey and Eli Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[148] The Clang Team. Matching the Clang AST, 2020. https://clang.llvm.org/docs/LibASTMatchers.html.

[149] Technology Services Group, University of Illinois at Urbana-Champaign. Ssniper Social Security Scanner for Linux, 2014. https://github.com/racooper/ssniper.

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

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

[152] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic Java Application Partitioning. In *European conference on object-oriented programming*, pages 178–204. Springer, 2002.

[153] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, page 9. ACM, 2014.

[154] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. 2013.

[155] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.

[156] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 683–693, 2017.

[157] Benjamin Venelle, Jérémy Briffaut, Laurent Clévy, and Christian Toinard. Security enhanced java: Mandatory access control for the java virtual machine. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–7. IEEE, 2013.

[158] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *Proceedings on Privacy Enhancing Technologies*, 2019(3):370–388, 2019.

[159] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine*, 35(2):46–55, 1997.

[160] 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.

[161] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407, 2018.

[162] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[163] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213. ACM, 2018.

[164] Westley Weimer and George C Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476. Springer, 2005.

[165] Ian H Witten, Eibe Frank, and Mark A Hall. Practical machine learning tools and techniques. *Morgan Kaufmann*, page 578, 2005.

[166] Ludwig Wittgenstein. *Philosophical investigations*. John Wiley & Sons, 2009.

[167] Scott Wolchok, Owen S Hofmann, Nadia Heninger, Edward W Felten, J Alex Halderman, Christopher J Rossbach, Brent Waters, and Emmett Witchel. Defeating vanish with low-cost sybil attacks against large dhts. In *NDSS*, 2010.

[168] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng 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*, pages 257–268. IEEE Press, 2019.

[169] Jinbo Xiong, Zhiqiang Yao, Jianfeng Ma, Ximeng Liu, and Qi Li. A secure document self-destruction scheme: an abe approach. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 59–64. IEEE, 2013.

[170] Ke Xu, Yingjiu Li, and Robert H Deng. Iccdetector: Icc-based malware detection on Android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264, 2016.

[171] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):23–32, 2013.

[172] Yang Yang, Chenhao Tan, Zongtao Liu, Fei Wu, and Yueting Zhuang. Urban

dreams of migrants: A case study of migrant integration in shanghai. *arXiv preprint arXiv:1706.00682*, 2017.

[173] Mengmei Ye, Jonathan Sherman, Witawas Srisa-an, and Sheng Wei. Tzslicer: Security-aware dynamic program slicing for hardware isolation. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 17–24. IEEE, 2018.

[174] Yubico Company. Yubico PAM module, 2019. https://developers.yubico.com/yubico-pam/.

[175] Fengshun Yue, Guojun Wang, and Qin Liu. A secure self-destructing scheme for electronic data. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 651–658. IEEE, 2010.

[176] Lingfang Zeng, Zhan Shi, Shengjie Xu, and Dan Feng. Safevanish: An improved data self-destruction for protecting data privacy. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 521–528. IEEE, 2010.

[177] Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 888–901. ACM, 2017.

[178] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. Recdroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*, pages 128–139. IEEE Press, 2019.