Exploring Value Set Analysis for Binary Code Hardening and Vulnerability Detection

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Sanchuan Chen, B.E., M.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Dr. Zhiqiang Lin, Advisor

Dr. Yinqian Zhang, Co-Advisor

Dr. Michael D. Bond

Dr. Atanas Rountev

© Copyright by

Sanchuan Chen

2021

Abstract

Value set analysis (VSA) is a static binary program analysis technique, which over-approximates the set of possible values of each data object in the program at every program point. Value set analysis can be used in binary code hardening and bug detection and thus protect software systems, especially legacy software systems without source code access, against program bugs such as notorious memory corruption bugs.

In this dissertation, we explore value set analysis in three categories of software defenses - taint analysis, data race detection, and binary similarity analysis. Specifically, we first present SelectiveTaint, an efficient static binary rewriting based, selective taint analysis framework for binary executables. SelectiveTaint leverages value set analysis to conservatively determine whether an instruction operand needs to be tainted or not, and then selectively taints the instructions of interest. We show that SelectiveTaint is times faster than that of the state-of-the-art taint analysis framework.

We also leverage value set analysis in data race detection of Intel SGX enclave binary code. We observe that unlike a traditional data race which occurs non-deterministically, a data race in SGX can be controlled, via manipulating thread creation, enclave call making, and thread execution. We propose a static binary analysis framework that automatically detect the controlled data races in enclave binary. Particularly, we systematically identify the possible shared variables via value set analysis and then explore the concurrent enclave

calls from both intended and unintended thread interleavings to detect whether a data race can occur.

In addition to exploring value set analysis in taint analysis and data race detection, we present vDiff, a novel binary code search framework that uses an architecture-generalized value set as a signature to capture function semantics for binary similarity comparison. A prototype implementation of vDiff searches binary code across five architectures (viz, x86, x86-64, ARM, AArch64, and MIPS), which achieves higher accuracy.

Dedicated to all associated with The Ohio State University

Acknowledgments

I would like to thank my advisor Professor Zhiqiang Lin for his support in my research. His guidance is always inspiring which helps me a lot in my pursuit of PhD degree. I would also thank my co-advisor Professor Yinqian Zhang and my dissertation committee members Professor Michael David Bond and Professor Atanas Ivanov Rountev for their valuable suggestions in my dissertation writing.

I also enjoy my stay at Security Lab with many great friends, Guoxing Chen, Mohit Jangid, Xin Jin, Mengyuan Li, Wubing Wang, Haohuang Wen, Yuan Xiao, Xiaokuan Zhang, Qingchuan Zhao, Chaoshun Zuo, and they all helped me in various ways.

Last but not least, I would like to express my thanks to my wife, Mrs. Meng Wang for her support in writing this dissertation.

Vita

August 2021	PhD, Computer Science and Engineering, The Ohio State University, USA.
January 2014	M.E., Computer Software and Theory, University of Chinese Academy of Sciences, China.
June 2009	B.E., Computer Science and Technology, University of Science and Technology of China, China.

Publications

Research Publications

Sanchuan Chen, Zhiqiang Lin, Yinqian Zhang. "SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting". In *USENIX Security 2021*, Aug 2021.

Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, Ten H. Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution". In *EuroS&P 2019*, Jun 2019.

Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, XiaoFeng Wang. "Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses". In *AsiaCCS 2018*, Jun 2018.

Guoxing Chen & Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, Dongdai Lin. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In *Oakland 2018*, May 2018.

Yuan Xiao, Mengyuan Li, Sanchuan Chen, Yinqian Zhang. "Stacco: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves". In *CCS 2018*, Oct 2017.

Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with DEJA VU". In *AsiaCCS 2017*, Apr 2017.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Overview

Value set analysis (VSA) is a static binary analysis technique, which uses abstract interpretation to safely approximate the set of values of each data object at each program point. As VSA does not assume the symbol information or debugging information is available for the analyzed binary, VSA first needs to identify a set of data objects called *abstract location*. By modeling binary memory layout, VSA partitions the whole memory into three kinds disjoint *memory regions*, *i.e.*, global, stack, and heap memory region, which contains *abstract locations* identified by VSA. VSA assigns the memory region to each instruction operand either based on the instruction semantics (*e.g.,* an absolution addresses instruction operand $[0x800100]$ is assigned to global memory region), or conservatively through data flow analysis(*e.g.,* an instruction operand containing an data object allocated in a `malloc` function is assigned to heap memory region). For each abstract location, VSA computes a value set for its possible value. Conventionally, VSA represents a value set as 3-tuple, the elements of which are the range of offsets with respect to global, stack, and heap memory region. For instance, the offset 0x4 with respect to the current stack memory can be represented as $(global \rightarrow \perp, stack \rightarrow [0x4, 0x4], heap \rightarrow \perp)$, which can be further abbreviated as $(\perp, [0x4, 0x4], \perp)$.

VSA has been widely used in many important applications, such as software failure diagnosis [37] and binary reassembling [112]. DEEPVSA [37] uses VSA on execution traces recorded via hardware tracing to recover data flow for tracing down the root causes of software crashes. Although VSA can perform alias analysis based on value set of each abstract location, for the truncated traces generated software crashes, the lose of context and over-approximation done by VSA limit its precision, and thus DEEPVSA facilitates VSA with a neural network to improve the precision of alias analysis. Ramblr [112] explores VSA in the *symbolization* of reassembling binary programs when source code is not available. In binary reassembling, symbolization is the procedure that converts absolute addresses into corresponding symbols and Ramblr improves the symbolization via a localized VSA, in which an integer value is deemed to be a symbol when a deference of a pointer is dependent on that integer value.

As shown in these aforementioned applications, value set analysis is becoming an essential binary analysis technique as it can infer the possible values an abstract location holds without the corresponding source code. We expect it will be further used in more and more applications. We are particularly interested in exploring value set analysis in binary code hardening and bug detection, as they are relevant defenses against emerging security bugs. As binary code hardening is to analyze, modify binary code and thus protect binary code from exploits and bug detection performed directly on binary code generally needs to reason about the possible values in the binary, value set analysis can certainly kick in to help the analysis in these two applications.

In this thesis, we explore value set analysis in three categories of software defenses - *taint analysis*, *data race detection*, and also *binary similarity analysis*. Specifically, we explore value set analysis in the following aspects:

- **Exploring Value Set Analysis for Selective Instrumentation in Taint Analysis.** Chapter 3 presents SELECTIVETAINT, the first selective static-binary-rewriting-based taint analysis framework to instrument taint logic, largely mitigating the performance overhead incurred by earlier DBI-based approaches. SELECTIVETAINT leverages a conservative tainted instruction identification approach, which statically identifies the instructions that will never involve tainted memory or registers by using VSA and then conservatively taints the rest instructions.

- **Exploring Value Set Analysis for Data Race Detection in Intel SGX Enclave Binary.** Chapter 4 presents SGX-RACER, the first static binary analysis tool for data race detection in SGX programs, by systematically exploring the possible concurrent ecalls from both intended and unintended thread interleavings to determine whether a shared variable access can lead to a data race. We also present a set of enabling techniques with binary analysis, including shared variable analysis, lock variable analysis, synchronization primitive identification, and also a new lockset-based data race detection algorithm particularly for SGX programs.

- **Exploring Value Set Analysis for Capturing Program Semantics in Binary Similarity Analysis.** Chapter 5 presents VDIFF, a novel binary code cross search scheme that leverages value-set analysis to capture the semantics of functions in a form that is resilient to changes from architecture. VDIFF applies a key enabling algorithm to refine the collected value sets and convert them into vectors, from which to compute similarity metrics and rank the similarity of functions.

1.2 Exploring Value Set Analysis for Selective Instrumentation in Taint Analysis

One of the mostly used techniques in software security is dynamic taint analysis [81], also called dynamic data flow tracking (DDFT), which tracks the data flow of interest during program execution and has been widely used in many security applications, such as exploit detection [15, 38, 81, 82, 87, 90], information flow tracking [107, 125], malware analysis [61, 88, 121], and protocol reverse engineering [10, 102]. However, the implementation of taint analysis often has high performance overhead. For instance, a state-of-the-art dynamic taint analysis framework `libdft` [54] imposes about 4X slowdown for `gzip` when compressing a file.

There has been a body of research that seeks to improve the performance of taint analysis. For instance, Jee et al. [49] applied compiler-like optimizations to eliminate redundant logic in taint analysis code. SHADOWREPLICA [48] improved the performance by decoupling taint logic from program logic, minimizing the information needed to communicate and optimizing the shared data structures between them. TAINTPIPE [74] explored a parallelism and pipeline scheme. STRAIGHTTAINT [73] combined an online execution state tracing and offline symbolic taint analysis for further performance improvement.

Interestingly, these general DDFT frameworks and their optimizations all built atop dynamic binary instrumentation (DBI), particularly Intel's PIN [70], to instrument the taint analysis logic at runtime. We believe a fundamental reason of using DBI for these frameworks is to basically avoid the code discovery challenge from static binary analysis. Note that PIN is a DBI tool, and it dynamically disassembles, compiles, and reassembles the executed code at runtime without any code discovery issues. The core module of PIN is a virtual machine (VM) that consists of a just-in-time (JIT) compiler, an emulator, and a

dispatcher. PIN also has a rich set of APIs used for Pintool's implementations. However, the VM and APIs both add additional performance overhead to a taint analysis tool.

Unlike DBI, static binary instrumentation (SBI) inserts the analysis code directly into the native binary and thus gets rid of the unnecessary DBI overhead incurred by such as JIT and emulation. Meanwhile, SBI would have fewer context switches, since the rewritten binary has a better code locality. While it is challenging to perform static binary analysis, recently there are substantial advancements in static binary rewriting and reassembling (e.g., UROBOROS [113], RAMBLR [112], MULTIVERSE [5], and recently Datalog Disassembly [33]). Therefore, it is worthwhile to revisit the taint analysis and study the feasibility of using static binary rewriting for more efficient taint analysis.

Another common ground that existing taint analysis frameworks share is that they all overly instrument the binary code at every possible instruction that can contribute the information flow, and they rely on the execution context to determine whether there is a need to taint the corresponding operand. However, if a static analysis could figure out precisely the instructions that will never get involved in taint analysis (e.g., via some conservative static analysis), it would have not instrumented them. Therefore, enabling taint analysis to selectively instrument the binary code statically is viable and highly desired.

We propose SELECTIVETAINT, an efficient selective taint analysis framework for binary code with static binary rewriting. There are two salient features in SELECTIVETAINT. First, it directly removes the overhead from dynamic binary translation, and is built atop SBI instead of DBI. Second, it scans taint sources of interest in the binary, statically determines whether an instruction operand will be involved in taint analysis by leveraging the value set analysis (VSA) [3, 4], and then selectively taints the instructions of interest. There are well-known challenges that SELECTIVETAINT must address, such as how to deal with

point-to (i.e., alias) analysis inside binary code. SELECTIVETAINT solves this problem by conservatively identifying the memory addresses that will never be involved in taint, and then taint the rest.

We have implemented SELECTIVETAINT atop SBI and evaluated it with a variety of applications consisting of the SPEC2006 CPUINT benchmark suite and network daemon programs such as Nginx web server. The evaluation results show that SELECTIVETAINT imposes significantly lower overhead than the current taint analysis framework such as `libdft`. We also confirm that SELECTIVETAINT can detect real-world exploits against the memory corruptions vulnerabilities in a variety of software including network daemon programs.

1.3 Exploring Value Set Analysis for Data Race Detection in Intel SGX Enclave Binary

Intel SGX allows programmers to protect application secrets in a hardware-isolated trusted execution environment (TEE), without trusting the system software such as operating systems and hypervisors. It provides the strongest security guarantee to an application to date: any memory reads or writes to an enclave from other software are prohibited regardless of their privileges. Unfortunately, Intel SGX is not absolutely secure, and it can still be attacked from a variety of sources including hardware (e.g., the VoltJockey attack [91], Rowhammer attack [47]), architectures (e.g., cache side channels [76]), operating systems (e.g., controlled side channels [9, 119], Iago attacks [13]), and applications from the enclave code itself (e.g., the buggy code to cause buffer overflow [60], use after free, double free, or null pointer deference [55]). In response to these attacks, numerous defenses have been proposed, particularly in defending against the side channel attacks with various approaches

including hardware transaction memory [36, 100], data location randomization [7], and oblivious memory primitives [96].

However, among the known defenses, none of them focuses on defeating one particular category of attacks—the controlled data races. Unlike traditional data race which occurs non-deterministically, a data race in SGX can be controlled. For instance, AsyncShock [116] has demonstrated that by manipulating the thread scheduling, an attacker is able to reliably exploit a TOCTOU bug to corrupt a shared variable from another thread between the check and the use. Unfortunately, the problem goes beyond what has been identified by AsyncShock since in SGX an intended single thread accessed variable can actually become a shared variable unexpectedly if a malicious OS creates an unintended attack thread to access it when the number of executed threads (i.e., `TCSnum`) for an enclave is configured to be more than one. While the programmers might have known this enclave re-entrancy attack, or at least to some extent, they may have failed to address it properly. Therefore, we must identify the controlled data races in SGX programs before being exploited.

While we could identify the data races from program source code, such an approach would depend too much on the programming languages used in the source code. Today, SGX enclave programs can be developed from a variety of programming languages, such as C/C++ with Intel SGX SDK, and Rust with Rust-SGX SDK. Thus, we may have to develop a detector for each language. Second, source code analysis could face the issues of what you see is not what you execute [4]. As such, a binary analysis based approach for detecting the races would be more appealing if that is possible. On the other hand, intuitively, to detect a data race it may require dynamic analysis to explore the thread interleavings. However, dynamic analysis is known for path coverage and scalability issues.

Therefore, we present SGX-RACER, a static binary analysis tool that systematically identifies the possible shared variables and explores both the intended and unintended thread interleavings in enclave code to inspect whether there are proper synchronizations on shared variables. A data race is identified if there is a lack of synchronization primitives when `TCSnum` is configured to be more than one. The key idea is to assume every ecall can run concurrently with another ecall (including itself), given a strong privileged attacker who can abuse enclave thread creation, ecall invocation, and fine-grained enclave code execution control. At a high level, SGX-RACER contains two phases of analysis: variable analysis phase and data race detection phase. The variable analysis phase recovers shared variables and lock variables from enclave code and generates locksets and lock acquisition histories. Data race detection phase considers each ecall to be possibly concurrent and performs a lockset-based [52] data race detection.

To detect data races in SGX enclave binaries, we still face three challenges: (1) identifying shared variables in the enclave binary, since identifying these variables needs comprehensive understanding of the binary instruction syntax and semantics; (2) identifying lock variables in the enclave binary which involves identifying various synchronization primitives used and even self-defined locks; (3) statically detecting data races in the face of concurrent ecall reentrancy, since enclave code can be called arbitrarily concurrent (when `TCSnum`>1) which introduces much more possible thread interleavings than a traditional computing environment. To solve these challenges, we have designed a set of enabling techniques including shared variable analysis, lock variable analysis, and synchronization primitive identification atop data flow analysis with enclave binaries. We also propose a lockset-based data race detection algorithm which assumes every ecall to be possibly concurrent.

We implement SGX-RACER atop binary analysis framework `angr` [112]. Being a binary analysis based solution, it allows SGX-RACER to analyze a variety of SGX binaries developed from different programming languages such as C/C++, and Rust. We evaluate SGX-RACER with open source SGX projects crawled from `github.com` and three popular SGX SDKs, namely, Intel SGX SDK, Open Enclave SDK, and Rust-SGX SDK.

1.4 Exploring Value Set Analysis for Capturing Program Semantics in Binary Similarity Analysis

Identifying binary code that is semantically equivalent but syntactically different across different machine architectures, platforms, and compilers (including their optimization levels) is crucial in many security applications, such as vulnerability and bug search [18, 86], malware detection [57], clustering [40], lineage tracing [46, 65], patch analysis [32, 53, 120], and exploit generation [8]. This problem has become increasingly important for securing the Internet of Things (IoT), due to reused (often vulnerable) code being recompiled and redeployed across a wider range of machine architectures (*e.g.,* x86, ARM, and MIPS). Defenders often lack access to source code for these deployments, making it difficult to find all instances of a vulnerability once vulnerable binary code on one architecture has been identified.

However, binary code clone search is nontrivial, since multiple factors can diversify even binaries compiled from the same source code. In particular, at the application level, compilers often perform aggressive compiler optimizations to boost runtime performance; at the platform level, compiled binaries have different binary headers and sections, different system calls, and different application binary interfaces (ABIs); and at the architecture level,

compiled binaries use different instruction set architectures (ISAs), resulting in dissimilar instruction sequences compiled from a common source.

Modern solutions to the code clone detection problem are typically either structure-based or value-based. Structure-based approaches analyze code control-flow graphs and other static structural information for similarities, such as by computing *n*-grams, instruction mnemonics, and sub-graphs [56], or extended control-flow graphs (*e.g.,* Genius [31] and Gemini [118]). However, many aggressive, architecture-specific compiler optimizations can greatly alter this structural information, impeding the detection of cross-architectural, cross-platform, and cross-compiler/optimization clones.

Value-based approaches instead compare values stored by code into registers and memory to assess similarity. These offer potential promise for higher-accuracy architecture- clone detection, since code clones often compute the same values even if by vastly different instruction sequences. Earlier efforts (*e.g.,* VaPD [50] and BLEX [28]) have explored traced variable values as signatures. Unfortunately, these approaches suffer from the coverage limitations fundamental to dynamic approaches—not all paths can be explored, and not all binaries can be executed (*e.g.,* without a suitable environment or sensor input). While recent advances have explored hybrid value-based approaches that combine symbolic execution with program verification [16] or theorem proving [71], these suffer from scalability issues.

As a counterpoint to this ongoing research, this paper investigates a purely static value-based approach to cross-architecture binary code clone detection, which has remained surprisingly unexplored despite the increasing attention in the literature. A purely static approach dodges the coverage problems faced by dynamic and hybrid approaches, potentially making it more scalable in practice. However, it raises the key challenge of how to compute meaningful variable values without resorting to low-coverage dynamic simulations. To solve

this, we draw upon recent advances in *value set analysis* (VSA) [3], which is playing an increasingly central role in binary analysis tools (*e.g.,* angr [101]) and can achieve high accuracy through deep learning (*e.g.,* DeepVSA [37]). These recent innovations raise the intriguing possibility of leveraging VSA for cross-architecture binary code search, which we here explore.

The result of our investigation is vDIFF, a new binary clone cross-search approach using VSA. Our key observation is that most functions compute value sets that show little variation across architecture and platform changes and compiler optimizations. However, using VSA for binary code search is by no means trivial. vDIFF must address numerous challenges, such as uniting differing binary code syntaxes across different architectures, identifying and eliminating architecture-specific value sets, and effectively comparing the refined value sets with proper similarity metrics to expressively measure the similarity. We show that many of these challenges can be addressed through a number of key insights. For example, many architectural differences can be linked to differences in memory layout; removing all the memory addresses from the results of VSA can therefore neutralize many architectural side effects that otherwise impair similarity detection.

We have implemented a prototype of vDIFF, which currently supports five different architectures (x86, x86-64, ARM, AArch64, and MIPS). At a high level, vDIFF first runs VSA over each function to generate value sets, which are then refined by removing the memory addresses and converted into a vector. Next, similarity metrics are computed for every function search pair. Finally, the functions are sorted and ranked as similarity candidates. We demonstrate the effectiveness of vDIFF by performing cross-architecture,

cross-platform, and cross-compiler/optimization code search on a set of benchmark programs. The evaluation results demonstrate that VDIFF correctly ranks similar functions with a high accuracy, especially when the targeted function has a large number of value sets.

Chapter 2: Background

2.1 Value Set Anlaysis

Value set analysis (VSA) [3,4] is a static program analysis technique. It over-approximates the set of possible values that each data object of the program could hold at each program point, and it uses a *value set* to represent the set of memory addresses and numeric value quantities.

Memory regions and abstract locations. VSA uses an abstract memory model that separates the address space into multiple disjoint areas that are referred to as *memory regions*. Memory regions in VSA consist of: a *global region* for memory locations storing uninitialized and initialized global variables, a *stack region* per function for memory locations of activation record of a procedure, and a *heap region* per heap allocation for memory locations allocated by a particular `malloc`-type of function call site. An *abstract location*, i.e., an *a-loc*, is a variable-like entity which spans from one statically known location to next statically known location, exclusively.

Abstract addresses and value sets. An *abstract address* in VSA is represented by a pair (memory-region, offset). A set of abstract addresses, i.e., a *value set*, can be represented using:

$$\{i|rgn_i \mapsto \{o_1^i, o_2^i, \ldots, o_{n_i}^i\}\}$$

More specifically, when there are at most one stack memory region and one heap memory region, the value set can be specified as 3-tuple [4]:

$$(global \mapsto O^g, stack \mapsto O^s, heap \mapsto O^h)$$

abbreviated as (O^g, O^s, O^h). A set of memory offsets in each memory region is represented by a *strided-interval* (SI): $s[l,u]$ where s is the stride, l and u are lower bound and upper bound. For instance, $(\{1,3,5\}, \perp, \perp)$ could be represented using SI as $(2[1,5], \perp, \perp)$.

The analysis is performed on a control-flow graph (CFG) in which each node represents an instruction (not a basic block as VSA is calculated for each instruction) and each edge represents a control flow transfer. A transfer function that characterizes the instruction semantics is associated with each edge. Note that since the address values and numeric values are interleaved in the binary, VSA tracks address values and numeric values at the same time.

2.2 Taint Analysis

Taint analysis is the process of tracking the flow of data of interest as they propagate during the program execution. It is also referred as data flow tracking (DFT) or information flow tracking (IFT). Static taint analysis (STA) is performed at compile-time without executing the program and could reason about all possible paths [2, 94]. Dynamic taint analysis (DTA) [81], also known as dynamic data flow tracking (DDFT), tracks the taint propagation at run-time and is more precise. DTA is usually implemented using virtualization or DBI. It can be performed per-process [54] or system-wide [121].

Taint tags are markings associated to registers and memory to indicate their taint status. Taint tags can have different granularities and sizes. A specific taint analysis could use a tag granularity at bit, byte, page, or file granularity. A finer granularity enhances taint analysis

```
1  void process(int client_sock, char *buffer, int size)
2  {
3          char ch;
4          int read_size = recv(client_sock, buffer, 2048, 0);
5          if(read_size > 0)
6          {
7                  ch = buffer[0];
8                  if(ch >= 'a' && ch <= 'z')
9                          buffer[0] = ch -32;
10                 write(client_sock, buffer, read_size);
11                 memset(buffer, 0, 1024);
12         }
13 }
14
15 int server(int client_sock)
16 {
17         int i = 0;
18         char buffer[1024] = {0};
19         for(i = 0; i < 3; i++)
20         {
21                 process(client_sock, buffer, 1024);
22         }
23         return 0;
24 }
```

Figure 2.1: A taint propagation example.

precision but adds performance costs, e.g., the storage cost for tag-related data structure , whereas a coarse granularity offers less precision but better performance. Tag size can be a single bit showing whether the corresponding registers or memory location is tainted or not, or can also be multiple bits or bytes showing more tag information, such as which part of the register or memory location is tainted by which part of the input (e.g., a particular byte offset).

A taint analysis typically consists of three components: taint sources, taint propagation, and taint sinks. In the following, we use a simplified networking program illustrated in Figure 5.1a, as a running example, to demonstrate how a typical taint analysis works.

- **Taint sources.** Taint sources are program points or memory locations where data of interest is introduced. Typically, taint analysis is interested in user input coming from locally or remotely. For example, in Figure 5.1a, if we are interested in the remote input, we will taint the data right after entering the system when calling `libc` function `recv` at line 4 with the data stored in `buffer`.

- **Taint propagation.** Taint tags are propagated during the program execution according to the taint propagation rules, which are specified with respect to the semantics of each instruction, e.g., the specific operands in the instruction, and also the side-effect of the instruction. For instance, for instruction `ADD src, dst`, a taint propagation rule could specify that the new tag of `dst` is a bit-wise `OR` of the tags of `src` and `dst`. In Figure 5.1a, at line 7 `ch` is assigned the tainted data of `buffer[0]` and at line 9 `buffer[0]` is calculated based on tainted `ch` which has a data dependency, whereas at lines 8-9 whether `buffer[0]` is assigned or not depends on the outcome of the predicate in the `if` statement, which involves a tainted `ch` with a control dependence between `buffer[0]` and `ch`. Note that most of the DDFT efforts (e.g., [48, 49, 54, 73, 74]) only consider taint propagation based on data dependencies.

- **Taint sinks.** Taint sinks are specific program instructions where taint analysis checks the existence of taint tags of interest for various security applications such as detecting control flow hijacks or information flow leakage. Common taint sinks are control flow transfer instructions. In Figure 5.1a, line 10 is a taint sink, invoking `libc` function

`write` that writes the content starting at `buffer` to `client_socket`. A user defined taint policy could check whether the content sending to network is tainted (to detect whether there is any secret data leaked to the client for instance).

2.3 Binary Instrumentation

Binary instrumentation is the process of instrumenting binary with additional analysis code added and meanwhile maintaining the original functionality. It is a widely used technique for many important security applications such as malware analysis and binary code hardening. Binary instrumentation could be either static or dynamic.

Static binary rewriting. Static binary instrumentation (SBI), also known as static binary rewriting, modifies the binary file directly. Static binary rewriting can be performed in three ways [112]: (1) trampoline-based, (2) lifting and recompiling, (3) symbolization and reassembling. Specifically, in trampoline-based approaches, hooks which detours the control flow to trampolines are added to the binary. In contract, for lifting and recompiling, the binary code will be first lifted into an intermediate representation (IR), then inserted with the code of interest in the IR, and finally compiled back. The first two approaches have been known in the community for years. Recently, symbolization and reassembling approach was proposed, in which a rewriter needs to identify the locations pointed by memory references first, and then symbolize those references. The process of converting numeric references back to symbols is called symbolization. After symbolization, the rewriter could correctly relocate binary in reassembling. The first two approaches impose significant overhead and the last approach may mix code with data and may not correctly separate them.

Dynamic binary instrumentation. Dynamic binary instrumentation (DBI) recovers the code while program is executing, which can correctly separate program code from data.

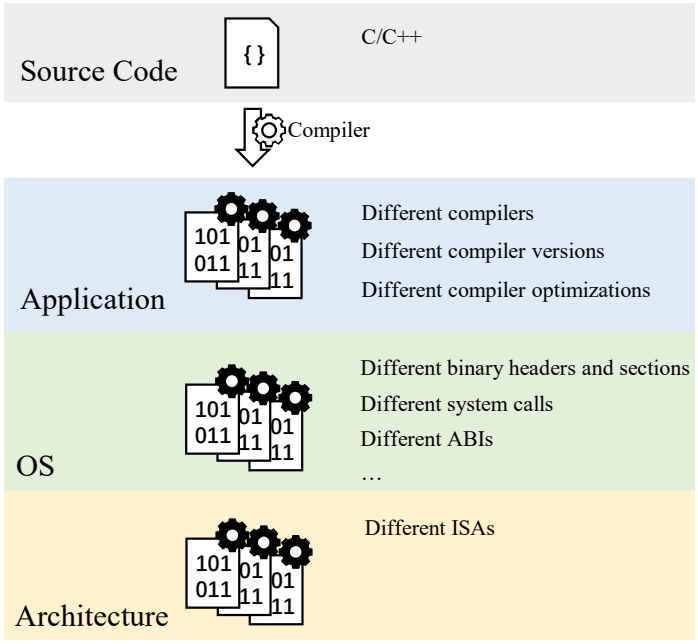


Figure 2.2: Different binaries compiled from the same source code

However, compared with static approaches, DBI has high performance overhead. There are generally two ways to implement DBI: using a trampoline, or using just-in-time (JIT) compiling. The trampoline approach replaces the instruction with a trampoline at run-time which jumps to the instrumented analysis code, and the JIT compiling approach dynamically compiles the binary on the fly.

2.4 Intel Software Guard Extensions (SGX)

Software Guard eXtensions (SGX) [41, 42] is a hardware feature in Intel x86/64 CPUs (since Skylake) to protect the confidentiality and integrity of application code and data, even when privileged system software (e.g., OS) are compromised. The trusted component of SGX application is executed inside an *enclave*, which is located in a protected memory region called enclave page cache (EPC). Enclave code can access EPC and the memory

outside enclave, but memory accesses to EPC from outside enclave are prohibited. A memory encryption engine inside the processor encrypts and decrypts memory EPC cache lines when they are written to and fetched from memory. There are two types of function calls between trusted component and untrusted component: (1) entering enclave call (ecall), through which the untrusted component makes an explicit call into an enclave, and (2) outside of enclave call (ocall), by which trusted component calls untrusted functions outside.

2.5 Threading Support in Intel SGX

Multiple threads can be executed concurrently inside an enclave. Each thread has a thread control structure (TCS) inside enclave, which contains control fields such as the thread's execution flag, the number of TCS (i.e., `TCSnum`), and the max number of TCS (i.e., `TCSMaxNum`), etc., specified in the configuration file. In addition, each thread has its own thread local storage (TLS). A TCS page, stack, and thread local storage variables make up the trusted execution context of a thread inside an enclave, whose state is saved to state save area (SSA) when CPU encounters a hardware exception to avoid trusted execution context revealed to untrusted software outside enclave.

However, thread creation inside enclave is not supported in SGX. A thread is first created outside the enclave and then bound to a trusted thread execution context inside enclave. The binding is controlled by untrusted code outside enclave and an enclave may have one of the two binding policies: (1) Non-binding mode. Any available trusted thread context is selected for an untrusted thread when a root call is made. "*A root call is defined as an enclave call that is not nested within another enclave call (or does not occur within the context of an enclave out call)*" [41, 42]. The same context is used for any nested enclave calls and all thread local storage variables are reinitialized when a root call is made. (2)

Binding mode. The same trusted thread context is bound to an untrusted thread when a root call is made and all thread local storage variables are not reinitialized for each root call.

A variety of SGX SDKs, e.g., Intel's SGX SDK [43], Microsoft's Open Enclave SDK [72], and Baidu's Rust-SGX SDK [25], all provide synchronization primitives (e.g., `sgx_spin_lock`, and `sgx_thread_mutex_lock` in Intel SGX SDK) to support multi threading inside enclaves. (A summary of the synchronization primitives and API functions provided by these SGX SDKs can be found in Table 2.1. Each SGX SDK also allows enclave developers to define TCS-related parameters in enclave configuration file, e.g., `TCSNum`, `TCSMaxNum`, the minimum number of available TCS during the lifecycle of an enclave `TCSMinPool`, and TCS binding policy `TCSPolicy`.

SGX SDKs	Sync. Primitive	Function
Intel SGX SDK	Spinlock	sgx_spin_lock sgx_spin_unlock
	Mutex	sgx_thread_mutex_lock sgx_thread_mutex_trylock sgx_thread_mutex_unlock
	Condition Variable	sgx_thread_cond_wait sgx_thread_cond_signal sgx_thread_cond_broadcast
Open Enclave SDK	Thread-once	oe_pthread_once
	Spinlock	oe_pthread_spin_lock oe_pthread_spin_unlock
	Mutex	oe_pthread_mutex_lock oe_pthread_mutex_trylock oe_pthread_mutex_unlock
	Read-write Lock	oe_pthread_rwlock_rdlock oe_pthread_rwlock_wrlock oe_pthread_rwlock_unlock
	Condition Variable	oe_pthread_cond_wait oe_pthread_cond_signal oe_pthread_cond_broadcast
Rust-SGX SDK	Thread-once	Once::call_once Once::call_once_force
	Barrier	Barrier::wait
	Spinlock	SgxThreadSpinlock::lock SgxThreadSpinlock::unlock
	Mutex	SgxThreadMutex::lock SgxThreadMutex::trylock SgxThreadMutex::unlock SgxThreadMutex::unlock_lazy
	Reentrant Mutex	SgxReentrantThreadMutex::lock SgxReentrantThreadMutex::trylock SgxReentrantThreadMutex::unlock
	Read-write Lock	SgxThreadRwLock::read SgxThreadRwLock::try_read SgxThreadRwLock::write SgxThreadRwLock::try_write SgxThreadRwLock::read_unlock SgxThreadRwLock::write_unlock
	Condition Variable	SgxThreadCondvar::wait SgxThreadCondvar::wait_timeout SgxThreadCondvar::signal SgxThreadCondvar::broadcast SgxThreadCondvar::notify_one SgxThreadCondvar::notify_all

Table 2.1: Synchronization primitives and functions in SGX SDKs

2.6 Concurrency Vulnerabilities

A concurrency vulnerability can occur due to multi-threading (which is enabled by multi-core computer architectures). A variety of concurrency vulnerabilities have been discovered over the years, such as data race [14, 26, 29, 80, 83, 97], atomicity violation [67, 69], and dead locks [29]. These vulnerabilities are prevalent in multi-threaded programs [68] and are notoriously hard to detect due to their non-deterministic natures.

Particularly, a data race occurs in a multi-threaded program if two threads access the same memory location without considering the ordering constraints between the two accesses, and also at least one access is a write. Moreover, a data race is challenging to be observed since a program can exhibit different behaviors when repeated even with the same input. Furthermore, a data race often quietly violates programmer's intention without causing a crash and could be noticed much later from the root cause.

To detect a data race, there are two typical approaches: (1) lockset based approach [97], which detects a data race if two threads access a memory location without holding a common lock, and (2) happens-before based approach [26], which detects a data race if two accesses from different threads are not ordered based on Lamport's happens-before relation [59].

2.7 Controlled Data Race Attack in Enclave Code

Controlled data race attack can be launched when `TCSnum` is greater than one. At a high level, there are three steps to carry out this attack as illustrated in Figure 2.3: **Step** ①–the attacker creates an untrusted thread which will be bound to the context of trusted enclave threads later, **Step** ②–the attacker makes ecalls in each created thread at his/her choice to enable concurrent execution of trusted enclave threads, and **Step** ③–the attacker forces the

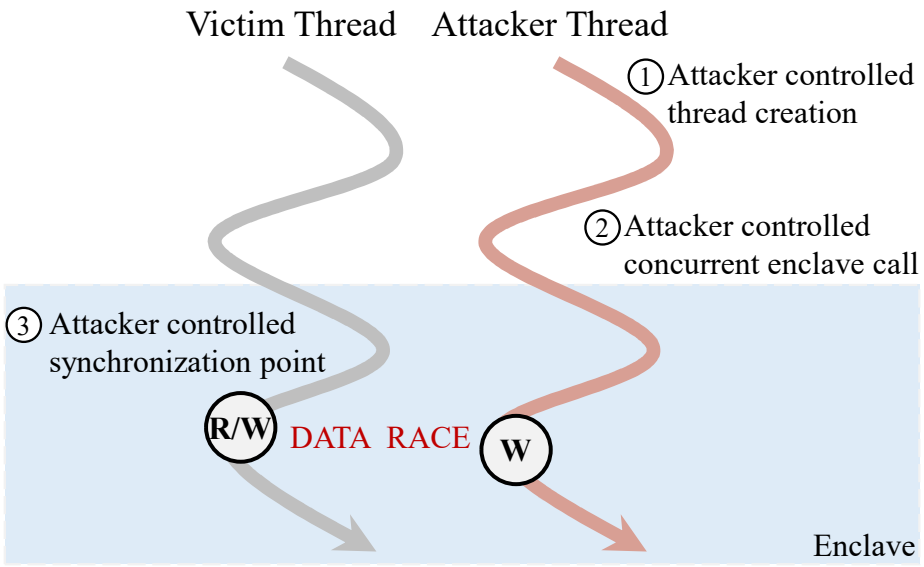


Figure 2.3: Illustration of controlled data race attacks

occurrence of a *controlled* data race by making the concurrent threads to synchronize at specific program points via interrupts or (controlled) page faults for example.

Therefore, the exploited data race is actually different from traditional data races, since they can be controlled in a deterministic manner by a malicious OS. They are extremely dangerous for the following three reasons. First, thread creation in SGX is not supported by the enclave itself, and instead it is controlled by OS. This is due to the reason that enclave applications are partitioned in a way [42, 64] that the size of its trusted computing base (TCB) is minimized and thread management is out of its TCB. As such, it provides a malicious OS the capabilities to create an arbitrary number of threads to attack a victim process. Second, an enclave call (`ecall`) has no ordering guarantee [42] and it can be called in an arbitrary order. A malicious OS can deliberately create multiple threads to trigger the same ecall, so that any instruction accessing global or shared heap variables may become reentrant, potentially causing data races. Third, unlike in traditional settings, an

enclave thread execution can be precisely controlled by a malicious OS using either page faults [119] or APIC timer interrupts [108, 109], The attacker could even cause single-step enclave execution at the granularity of instruction level, which leaves ample room for them to synchronize two threads precisely at any point of their execution and thus cause *controlled* data races.

The root cause of controlled data races is that a portion of the enclave code is non-reentrant. Unlike traditional programs, enclave programs are under the management of an untrusted OS, which controls the thread creation, the entrance of ecalls, and the interrupt and resumption of execution at any point. A controlled data race attack could also target an intended single-threaded enclave program, if the malicious OS can create multiple threads and allow each of the threads to instantiate the single-threaded enclave code (e.g., via calling ecalls). In this way, any global variable used in a single-threaded enclave program now accidentally becomes shared by these concurrent threads when `TCSnum`>1. It is likely that many developers could make such mistakes.

2.8 Root Causes of Binary Code Differences

There could be numerous factors that can contribute to the differences of the final binary code. Based on how a program is compiled and executed, we can summarize these factors into three categories as shown in Figure 2.2, from application level, to operating system level, and to architecture level.

Application level. Program source code needs to be first compiled. Different compilers may have different code generation strategies, and even for the same compiler, it can also use different optimization levels. For instance, compilers typically perform a series of optimization transformations that produce semantics-equivalent code yet consuming

fewer resources or running faster. Compiler optimizations span different levels and typical optimizations include but not limited to: (1) *peephole optimizations* that inspect multiple adjacent instructions and replace them with a short and faster sequence of instructions; (2) *local optimizations* that are fast optimizations only examining instructions within basic block boundaries; (3) *global optimizations* that operate on whole functions, where function calls and global variable accesses may occur; (4) *loop optimizations* that examine the instructions within loops, which could improve cache performance and parallelism; (5) *inter-procedural optimizations* that consider all instructions and generally take place at link time.

Operating system level. Different operating system platforms (*e.g.,* Linux and Windows) have different binary loaders and thus use different *binary file formats* (*e.g.,* ELF format in Linux and PE format in Windows). These formats contain data structures to help binary loaders to manage the executable code, which generally consist of multiple headers and sections. Typical sections are *.text* and *.data*, which contain program code loaded with executable/read permissions and global data loaded with non-executable/read/write permissions. Besides the data structure differences in different binary file formats, the main platform dependencies come from application binary interface (ABI) and system calls [58]. For instance, Linux dispatches system call through instruction `INT 0x80`, or virtual dynamic shared object (VDSO), whereas Windows system calls are made by calling APIs, such as `NtCreateFile` and `NtOpenFile`.

Architecture level. Semantics-similar source code or even the same source code can have vastly different binary code on different architectures. There are many reasons to cause the differences, and these include: (1) Reduced instruction set computer (RISC) vs. complex instruction set computer (CISC). RISC architectures embrace the philosophy of having a few simple and general instructions instead of a large number of complex and specialized

instructions as in CISC architectures. RISC architectures are also load-store architectures, in which binary code tends to contain multiple `load` and `store` instructions. For the five architectures supported by VDIFF, x86 and x86-64 are CISC architectures and the other three are RISC architectures. (2) Registers. Each architecture offers different kinds and number of registers, *e.g.,* ARM has 30 general purpose registers while x86 has just 8 32-bit general purpose registers, and ARM also has a link register that stores the return address whereas x86 does not have a peer link register. (3) Instruction opcode. For instance, ARM has the `BL` instruction (branch with link) which calls subroutine and saves the return address in the link register, while x86 uses the `call` instruction which stores the return address on the stack. (4) Instruction operands. For example, the `ADD` instruction takes three operands in ARM but only takes two operands in x86. (5) Word size. ARM, x86, and MIPS have a 32-bit word size while AArch64 and x86-64 have a 64-bit word size. (6) Endianness. For instance, x86 is little-endian, and MIPS is bi-endian (it can also be configured in either big or little endian).

Chapter 3: Exploring Value Set Analysis for Selective Instrumentation in Taint Analysis

3.1 Motivation

Taint analysis has been widely used in many security applications such as exploit detection, information flow tracking, malware analysis, and protocol reverse engineering. State-of-the-art taint analysis tools are usually built atop dynamic binary instrumentation, which instruments at every possible instruction, and rely on runtime information to decide whether a particular instruction involves taint or not, thereby usually having high performance overhead. This paper presents SELECTIVETAINT, an efficient static binary rewriting based, selective taint analysis framework for binary executables. The key idea is to use static binary rewriting instead of dynamic binary instrumentation to selectively instrument the instructions involving taint analysis. At a high level, SELECTIVETAINT scans taint sources of interest in the binary, leverages value set analysis to conservatively determine whether an instruction operand needs to be tainted or not, and then selectively taints the instructions of interest.

3.2 Challenges

To clearly illustrate the challenges of selective taint analysis, we still use the example code shown in Figure 5.1a. This program receives three messages from a client (line 19-22), capitalizes the first character in each message if needed (line 8-9), and sends the messages back to the client (line 10). It has a buffer overflow vulnerability at line 4, when receiving the input with size larger than 1024 bytes. The taint source of our interest is the network input stored in array `buffer`, which is tainted by `libc` function `recv`. The taint sink of our interest is the control flow transfer instruction `ret` of function `server` at line 23 of Figure 5.1a, assume our objective is to detect the control flow hijacks.

Performing selective binary code taint analysis using static binary rewriting is by no means trivial. Unlike DBI-based approaches where taint analysis logic is instrumented at runtime, a SBI-based approach has to analyze and rewrite the binary statically. In addition to the challenges from static binary disassembling and rewriting (they are orthogonal to our problem and they should be solved separately), SELECTIVETAINT has to address at least the following unique challenge—how to determine whether a disassembled instruction needs to be instrumented by taint analysis. If so, we will rewrite it accordingly based on the taint semantics (e.g., whether this instruction introduces a taint sources, contributes to taint propagation, or it is a taint sink).

Essentially, the problem becomes how to determine the taintedness of an instruction according to its operands including both memory addresses and registers without executing the binary. Determining the taintedness of registers is easier compared to memory addresses, since registers can be directly identified based on names whereas a memory address cannot be easily resolved. Therefore, determining the taintedness for memory addresses is much harder in SBI. More specifically, different from DTA in which a memory address has a single

runtime address at each program point, static binary taint analysis can only conservatively infer the possible values for a symbolic memory address at each program point. Except for global memory addresses, symbolic addresses of stack and heap are only in relative addresses when performing the static analysis. In addition, there are also unknown inputs (from a command line, local files and keystrokes, or remote network packet) that also make the problem hard.

3.3 A Running Example

It is obvious that in order to address the aforementioned challenges, it requires the inference of possible values of both registers and memory cells at each program point. Fortunately, a key technique in this direction is the VSA [3, 4], which seeks to compute the possible values at each symbolic memory address and register. Therefore, with VSA, we could determine whether a particular memory address or register involves taint or not, e.g., whether it is an alias to the address of our interest, or it will hold the propagations of the tainted data.

To see exactly how VSA helps our analysis, we show the value set analysis results of our running example along with its assembly code in Table 3.1. At the beginning of function `server`, the initial `esp` has a value set of ($\perp$, 0x0, $\perp$). After executing `push` instruction at `0x8048687`, `esp` has a value set of ($\perp$, -0x4, $\perp$). The analysis continues, and computes the rest of the VSA for each register and memory operand. Now with the computed VSA, we can easily see that `ebx` at `0x80486a9` and `eax` at `0x80486c6` have the same value sets ($\perp$, -0x410, $\perp$), and thus these two registers are actually aliased. In fact, both of them refer to the address of the local variable `buffer` defined in function `server`.

Assembly		Value Set Examples
<server>:		
8048687	push %ebp	esp:(⊥,-0x4,⊥)
8048688	mov %esp,%ebp	ebp:(⊥,-0x4,⊥)
804868a	push %edi	
804868b	push %ebx	
804868c	sub $0x420,%esp	esp:(⊥,-0x42c,⊥)
8048692	movl $0x0,-0xc(%ebp)	
8048699	lea -0x40c(%ebp),%ebx	
804869f	mov $0x0,%eax	
80486a4	mov $0x100,%edx	
80486a9	mov %ebx,%edi	ebx:(⊥,-0x410,⊥)
80486ab	mov %edx,%ecx	
80486ad	rep stos %eax,%es:(%edi)	
80486af	movl $0x0,-0xc(%ebp)	
80486b6	jmp 80486d9	
80486b8	movl $0x400,0x8(%esp)	
80486c0	lea -0x40c(%ebp),%eax	
80486c6	mov %eax,0x4(%esp)	eax:(⊥,-0x410,⊥)
80486ca	mov 0x8(%ebp),%eax	
80486cd	mov %eax,(%esp)	
80486d0	call 80485fd<process>	
80486d5	addl $0x1,-0xc(%ebp)	
80486d9	cmpl $0x2,-0xc(%ebp)	
80486dd	jle 80486b8	
80486df	mov $0x0,%eax	
80486e4	add $0x420,%esp	
80486ea	pop %ebx	
80486eb	pop %edi	
80486ec	pop %ebp	
80486ed	ret	

Assembly		Value Set Examples
<process>:		
80485fd	push %ebp	ebp:(⊥,-0x434,⊥)
80485fe	mov %esp,%ebp	ebp:(⊥,-0x434,⊥)
8048600	sub $0x28,%esp	esp: (⊥,-0x45c,⊥)
8048603	movl $0x0,0xc(%esp)	
804860b	movl $0x800,0x8(%esp)	buffer size:(0x800,⊥,⊥)
8048613	mov 0xc(%ebp),%eax	
8048616	mov %eax,0x4(%esp)	buffer addr:(⊥,-0x410,⊥)
804861a	mov 0x8(%ebp),%eax	
804861d	mov %eax,(%esp)	
8048620	call 80484f0<recv@plt>	$\mathscr{V}_u$= $\mathscr{S}$ - (⊥,[-0x410,0x3f0],⊥)
8048625	mov %eax,-0xc(%ebp)	
8048628	cmpl $0x0,-0xc(%ebp)	
804862c	jle 8048685	
804862e	mov 0xc(%ebp),%eax	
8048631	movzbl (%eax),%eax	
8048634	mov %al,-0xd(%ebp)	
8048637	cmpb $0x60,-0xd(%ebp)	
804863b	jle 8048651	
804863d	cmpb $0x7a,-0xd(%ebp)	
8048641	jg 8048651	
8048643	movzbl -0xd(%ebp),%eax	
8048647	sub $0x20,%eax	
804864a	mov %eax,%edx	
804864c	mov 0xc(%ebp),%eax	
804864f	mov %dl,(%eax)	
8048651	mov -0xc(%ebp),%eax	
8048654	mov %eax,0x8(%esp)	
8048658	mov 0xc(%ebp),%eax	inst. is tainted, as (⊥,-0x410,⊥) $\not\subseteq \mathscr{V}_u$
804865b	mov %eax,0x4(%esp)	
804865f	mov 0x8(%ebp),%eax	
8048662	mov %eax,(%esp)	
8048665	call 80484a0<write@plt>	
804866a	movl $0x400,0x8(%esp)	
8048672	movl $0x0,0x4(%esp)	
804867a	mov 0xc(%ebp),%eax	
804867d	mov %eax,(%esp)	
8048680	call 80484c0<memset@plt>	
8048685	leave	
8048686	ret	

Table 3.1: The running example assembly code snippets.

To statically analyze which instructions need to be tainted, a straw-man approach is to statically maintain tainted value sets (i.e., value sets of registers and symbolic memory that need to be tainted) at each program point. In particular, this approach checks whether value sets of all operands in an instruction are subsets of the tainted value set, if so, this instruction is added into the tainted instruction set, the register or symbolic memory of the corresponding operand is added to the tainted value sets if the taint will be propagated to this operand, and the corresponding taint rule is instrumented to taint this instruction.

However, when analyzing real-world binaries, VSA may lose its precision due to various factors such as imprecise CFG, sophisticated static point-to analysis (which is an undecidable problem [93]), and unknown inputs. Consequently, as illustrated in Figure 3.1, we may not be able to get the ideal tainted instruction set $\mathscr{I}$ for the instructions that need to be tainted, and instead the VSA identified must-tainted instruction set $\mathscr{I}_t$ can have false negatives because of the imprecision mentioned. On the other hand, by using VSA, we can also identify must-not-tainted instruction set $\mathscr{I}_u$ that will never be involved in taint analysis. Therefore, in order not to have any false negatives (no missing of attacks) when using taint analysis, we eventually decide to taint the instructions that are not in the $\mathscr{I}_u$, though the worst case is we can taint all instructions similarly to all DBI-based taint analysis. Our key objective is to confidently enlarge $\mathscr{I}_u$ as much as possible.

As in our running example, in Table 3.1, the instructions in light gray are identified as in must-not-tainted instruction set $\mathscr{I}_u$, the instructions in dark gray are identified as in must-tainted instruction set $\mathscr{I}_t$, and all instructions not in light gray are our conservatively tainted instructions. For each instruction, a must-not-tainted value set $\mathscr{V}_u$ is maintained and if value sets of all operands in an instruction are subsets of $\mathscr{V}_u$, the instruction is added to $\mathscr{I}_u$. For instance, for instructions at `0x804861a` and `0x804861d` before taint introduction at `0x8048620`, must-not-tainted value set $\mathscr{V}_u$ equals value set $\mathscr{S}$, which contains all possible values at this execution point (recall VSA is flow sensitive analysis). At taint source `0x8048620`, must-not-tainted value set $\mathscr{V}_u$ is updated by removing value set ($\perp$,[-0x410,0x3f0],$\perp$) from $\mathscr{V}_u$, as the tainted buffer starts at ($\perp$, $-0x410$, $\perp$) with a buffer length `0x800`. At `0x8048658`, `[ebp+0xc]` has value set ($\perp$, -0x410, $\perp$), which is not a subset of must-not-tainted value set $\mathscr{V}_u$ and thus this instruction is not added to $\mathscr{I}_u$ and is instrumented. Eventually SELECTIVETAINT will conservatively taint all instructions

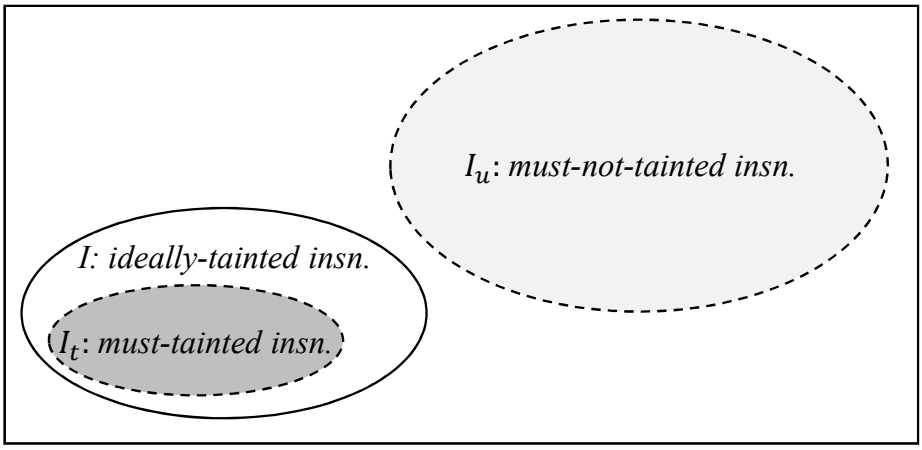


Figure 3.1: The Essence of SELECTIVETAINT.

not in $\mathscr{I}_u$, i.e., instructions not in light gray, which consists of all instructions in $\mathscr{I}_t$, i.e., instructions in dark gray, with five additional instructions in white.

Scope and Assumptions. In this work, we focus on x86 binaries with ELF format running atop Linux platform. We assume the binary code is not obfuscated, and we are able to get their correct disassembly. For proof-of-concept, we demonstrate the use of DTA to track the untrusted user input through static binary rewriting, and detect the memory exploits by just using a single bit (tainted or not) in our taint record. Also, our static binary rewriting is based on Dyninst [6]. While it is not perfect, it has been widely used in building many static binary rewriting-based prototypes, e.g., TypeArmor [111], BinArmor [103], PathArmor [110], MARX [84], and most recently UnTracer [79].

3.4 Detailed Design

In this section, we present the detailed design of SELECTIVETAINT. As illustrated in Figure 3.2, there are four key components inside:

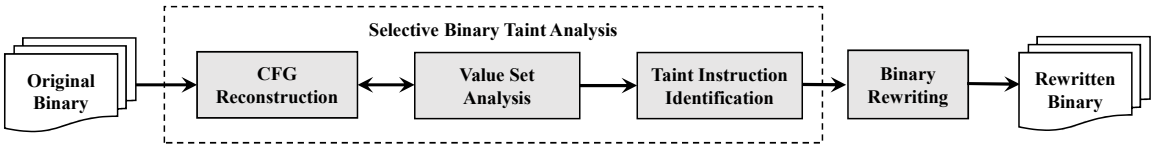


Figure 3.2: Overview of SELECTIVETAINT.

- **CFG Reconstruction (§3.4.1).** When given an application binary code, we will first rebuild its CFG starting from the `main` function. If there is a library call, we will resolve the calling target and use the function summaries to decide whether further instrumentation of the library is needed. If an indirect `jmp/call` is encountered, we will perform backward slicing [112] and use the VSA and type information to resolve the target.

- **Value Set Analysis (§3.4.2).** VSA [3] has become a standard technique in static binary analysis for determining the possible values of a register or a symbolic memory address. We use the VSA to help identify the instruction operands that are never involved in the taint analysis.

- **Taint Instruction Identification (§3.4.3).** Selective tainting essentially aims to identify the instructions that are involved in the taint analysis. With the identification of $\mathscr{I}_u$ by VSA, we then start from the instructions that introduce the taint sources, and systematically identify the rest of instructions that are not in $\mathscr{I}_u$.

- **Binary Rewriting (§3.4.4).** Once the instructions have been identified, we then use the static binary rewriting to insert the taint analysis logic including tracking of the taint sources and taint propagations as well as the taint checks at the taint sinks.

3.4.1 CFG Reconstruction

The first step of SELECTIVETAINT is to rebuild the CFG, when given an application binary. This is quite a standard process and the only additional challenge is to identify the control flow targets of the indirect calls and jumps, as they are important to compute the VSA. To get the CFG, we first reconstruct the possible control flow targets using Ramblr [112] approach, and in case of undecided target (e.g., `jmp/call eax`), we use the following approaches:

Handling Indirect Call. We adopt and implement two forward-edge CFI identification approaches, namely TypeArmor [111] and τCFI [77], to recover the type information (i.e., parameter count and parameter type) about actual and formal parameters at the callsites and callee functions. By connecting the matching callsites and callees regarding these type information, we build a CFG which is an over-approximation of actual CFG. The type information is generated by running liveness analysis at indirect callsites and use-def analysis at callees.

Handling Indirect Jump. We first use our implemented VSA to resolve the indirect jump target and connect the jump target if it is solved. Otherwise, we determine whether the function that contains the indirect jump uses any external data references (e.g., global variable addresses): if not, we connect all of the possible basic block starting address in this function as the potential jump target (we still consider it local); otherwise, we connect the jump target with all function entry addresses. The rationale is we notice all inter-procedural jumps we encountered are from compiler optimizations, and basically compiler optimizes the call instruction with an indirect jump. We therefore connect the indirect jump in this way to get an over-approximation of the CFG.

3.4.2 Value Set Analysis

Our VSA Algorithm. A key technique inside SELECTIVETAINT is the VSA [3], which is a context-sensitive and flow-sensitive whole program analysis. As described in algorithm 1, our `whole_program_VSA` first initializes the `ValueSet` for each instruction in the program with an initial `esp`, initial empty heap, and initial memory cell values resolved from original binary. Then, function `VSA` is called to analyze each function `func`, which is of work list style with multiple iterations on each individual instruction until no changes are discovered (fixed point is reached). The context and value sets are adjusted depending on the type of instruction opcode, e.g., for `call/ret` instruction, inter-procedural analysis is performed and the environment is adjusted accordingly including changing the current stack region and matching formal and actual parameters value set are carried out.

Practical Challenges. While the idea of calculating VSA is simple, it has a number of practical challenges when used for data flow tracking, such as context-sensitive, flow-sensitivity, and alias analysis. In the following, we describe these challenges and also how we have addressed them below:

(I) Handling context-sensitivity. It will be overly complicated if a function is called multiple times when performing the inter-procedureal analysis in a CFG. We therefore augment our VSA with a cloning-based context sensitivity analysis [117]. Basically, we have a separate analysis for each function clone per calling context. More specifically, we generate a function clone for every acyclic path through a program call graph and, for cyclic paths, we merge all functions in a strongly connected component to have a single function context for them as in [117].

Algorithm 1 Whole Program Value Set Analysis

Function *whole_program_VSA(CFG, ValueSet)*:
    input : control flow graph *CFG*, value set *ValueSet*
    output : ValueSet[i] for each instruction i
1   ValueSet, context ← init()
    VSA(entryFunc, context)
Function *VSA(CFG, ValueSet, func, context)*:
    input : control flow graph *CFG*, value set *ValueSet*, function *func*, *context*
    output : ValueSet[i] for each instruction i
2   worklist ← {entryInst}
    while *worklist* $\neq \emptyset$ **do**
3   i ← pop(worklist)
        if *callInst(i)* **then**
4   newContext ← adjustContext(context, callee(i))
            VSA(CFG, ValueSet, callee(i), newContext)
5   **if** *retInst(i)* **then**
6   adjustContext(context, caller(i))
7   **if** *condInst(i)* **then**
8           $\text{ValueSet}^i_{exit_n} \leftarrow \text{ValueSet}^i_{entry} \sqcap^{VS} \text{ValueSet}_{c_n}$
9   **else**
10   **if** *uninitialized*(op_i) **then**
11              $\text{ValueSet}^i_{exit}[\text{addr}(\text{op}_i)] \leftarrow (\top,\top,\top)$
12          $\text{newValueSet}^i_{exit} \leftarrow \boldsymbol{EXE}(i, \bigsqcup_{entry_n \in entry} \text{ValueSet}^i_{entry_n})$
            if $newValueSet^i_{exit} \neq ValueSet^i_{exit}$ **then**
13              $\text{ValueSet}^i_{exit} \leftarrow \text{ValueSet}^i_{exit} \sqcup \text{newValueSet}^i_{exit}$
                push(worklist, succs(i))

(II) Handling flow-sensitivity. Since VSA is flow-sensitive and per-instruction, it is an engineering challenge to inspect each instruction statically. We therefore borrow the idea of how symbolic execution interprets each instruction and updates the corresponding symbolic states. Essentially, when perform our flow sensitivity analysis, we need to interpret each instruction, and updates the VSA based on its semantics. Since symbolic execution is well studied (with many open source tools), we do not describe how we implement our interpreter and instead we abstract it as a simple *EXE* (line 18) in algorithm 1, which is responsible to capture the value set changes for each analyzed instruction (working as a transfer function in static analysis).

(III) Handling a-locs with unknown values or addresses. Performing VSA on binary suffers from the lack of dynamic information (e.g., calling context, and concrete memory addresses). One major issue when applying VSA on real-world binary is uninitialized variables and their aliases. Among these uninitialized variables, some are used in address calculation, which leads to a-locs with unknown addresses. To conservatively taint instructions, we need to infer the value set of these unknown addresses; otherwise the reads and writes to them would indicate the reads and writes to the whole address space.

In case VSA encounters an a-loc with uninitialized value or address due to system inputs for instance, the special handling is shown in lines 16-17 in algorithm 1. In particular, our analysis will assume the uninitialized a-loc to have any value, i.e., with the value set $(\top, \top, \top)$. The cases which VSA cannot determine the address are:

- **Unknown values from command line input (CLI), e.g.,** `argv[].` The `argv` elements are pointers which is uninitialized at analysis-time. As shown in Figure 3.3a, instruction at `0x8048745` reads `argv[1]` which is unknown at analysis-time.
- **Unknown addresses or values passed from missing callers.** Even we use approaches such as TypeArmor to recover CFG, there are still some callee functions without callers and the calling context is missing for these callee. As shown in Figure 3.3b, the function `Perl_do_open9` has no identified callers, and thus, the value of parameter at instruction `0x804e9bb` is uninitialized.
- **Unknown addresses or values due to library functions and system calls.** For instance, `fopen` function returns a pointer which is a pointer to `FILE` struct that is uninitialized at analysis-time as illustrated in Figure 3.3c.

```
429.mcf
080486d0 <main>:
 80486d0: push   %ebp
 80486d1: mov    %esp,%ebp
 ...
 8048730: mov    0xc(%ebp),%eax
 8048733: movl   $0xc8,0x8(%esp)
 804873b: movl   $0x989680,0x804e890
 8048745: mov    0x4(%eax),%eax
```

(a) Entry-function uninitialized variable

```
400.perlbench
0804e9b0 <Perl_do_open9>:
 804e9b0: sub    $0x3c,%esp
 804e9b3: lea    0x5c(%esp),%eax
 804e9b7: mov    %eax,0x1c(%esp)
 804e9bb: mov    0x58(%esp),%eax
 804e9bf: movl   $0x1,0x20(%esp)
 804e9c7: mov    %eax,0x18(%esp)
 804e9cb: mov    0x54(%esp),%eax
 804e9cf: mov    %eax,0x14(%esp)
 804e9d3: mov    0x50(%esp),%eax
 804e9d7: mov    %eax,0x10(%esp)
 804e9db: mov    0x4c(%esp),%eax
 804e9df: mov    %eax,0xc(%esp)
 804e9e3: mov    0x48(%esp),%eax
 804e9e7: mov    %eax,0x8(%esp)
 804e9eb: mov    0x44(%esp),%eax
 804e9ef: mov    %eax,0x4(%esp)
 804e9f3: mov    0x40(%esp),%eax
 804e9f7: mov    %eax,(%esp)
```

(b) Incomplete CFG caused uninitialized variable

```
429.mcf
08048fe0 <read_min>:
 8048fe0: push   %ebp
 8048fe1: push   %edi
 8048fe2: push   %esi
 8048fe3: push   %ebx
 8048fe4: sub    $0x12c,%esp
 ...
 8049010: call   8048670 <fopen@plt>
 8049015: test   %eax,%eax
 8049017: mov    %eax,0x38(%esp)
```

(c) `fopen` uninitialized variable

Figure 3.3: Uninitialized variable examples in whole program VSA.

3.4.3 Taint Instruction Identification

After our whole program VSA analysis, we next need to identify the instructions that need to be instrumented for the taint analysis with the computed VSA. To this end, we have to decide whether a memory address involves taint or not, which essentially leads to problem of point-to (i.e., alias) analysis. However, due to the imprecision with the static point-to analysis, we may not be able to resolve all memory addresses with VSA [3,4], and instead we focus on identifying the addresses that will never be involved in taint analysis for each specific instruction (since VSA is flow sensitive). Initially, all instructions will be marked tainted (i.e., they will all be instrumented for taint analysis). As described in §3.3, our key objective is to minimize this set, by identifying and enlarging the must-not tainted set. In the following, we describe how we achieve this.

3.4.3.1 Must-not Tainted Analysis

In order to statically identify instructions never involved in taint analysis, we should know the must-not tainted value set, which is an opposite, more conservative counter-part of the intuitive tainted value set, at each program point. This is also a data flow analysis problem, and we have to inspect each instruction to decide whether its operand will never be involved in taint or not.

Identification Policy. Must-not-tainted set is based on the following policy: (1) instructions unreachable from taint sources are *removed* from the must-not-taint set, e.g., the instructions at the beginning of the program to the first instruction that introduces the taint source (which is one of the big differences compared to DBI-based taint implementations); (2) instructions with operands of potentially tainted or unknown value sets are *removed* from must-not-taint set; (3) instructions whose operands hold literal values are added to must-not-taint set since

none of the operands will be tainted, e.g., instruction `inc eax` with `eax` containing a literal value is *added* to must-not-taint set; (4) instructions whose opcode indicates it will not be involved in taint propagation are *added* to must-not-taint set, e.g., control-flow instructions (e.g., `jmp`) and compare and test instructions (e.g., `cmp`, and `test`). The must-not tainted value set will propagate along with data flow, and it is a must analysis.

Resolving operand's addresses. To conservatively track the must-not tainted value sets, we have to look into different types of memory access of an instruction operand: (1) for constant memory address, e.g., `[0x8000200]`, we can easily infer that it is a global variable rather than local variable or heap variable and the must-not tainted value sets of that address can be updated based on the constant memory address, e.g., if this constant memory address may be tainted, the constant memory address would be removed from our must-not tainted set; (2) for a memory access based on `ESP` register, which we call *stack pointer addressing*, e.g., `[esp + 0x4]`, we identify it as a stack variable, the stack region and offset can be obtained through our whole program analysis caller/callee stack information; (3) for a memory access without `ESP` register, e.g., `[eax]`, this is tricky since we may not know whether it is a stack, global or heap variable; we thus use the VSA result to decide the value set of the memory access: if the VSA cannot decide whether the memory access address is tainted or not, we conservatively remove it from the must-not tainted set.

Resolving operand's values. Once the algorithm meets an instruction operand that is uninitialized (it can lead to an alias cannot be resolved), as mentioned in §3.4.2, we conservatively taint associated variables, depending on the specific cases:

- **Unknown value from CLI (e.g., Figure 3.3a).** Based on where the input value is going to be stored, we assign a corresponding uninitialized value for these variables. For instance, we will assign an uninitialized value for a stack varaible which belongs

to the stack of the caller of `main` and prior to the stack of `main` function and process the must-not analysis as usual.

- **Unknown value passed from missing callers (Figure 3.3b).** A caller function passes function parameters to a callee function, causing aliasing between actual parameters and formal parameters. When CFG reconstruction cannot determine the callers for a callee, it results in unknown value from the missing callers. We conservatively remove all memory access instructions in the function and all uses of these variables outside the function from must-not tainted set. To optimize our analysis, we do not over-taint all global variables, instead we taint the data based on their types (the type inference is described below) in global sections such as `.data` and `.bss`.
- **Unknown value due to library function calls and system calls (Figure 3.3c).** We taint these unknown variables according to the semantics of library functions and system calls. For instance, the pointer returned by `fopen` is put in must-not tainted set at the program point right after the library call and the pointer returned is assigned a value set in a special heap region.

Variable type inference. To taint instructions more precisely, we perform a simple variable type inference to determine whether a variable is a pointer or not. We care them because we want to identify the potential pointers that can hold the taint buffer. The analysis is based on whether a variable is dereferenced or whether it is a pointer type parameter or return value of known library functions as type sinks [63]. For instance, `movzbl (%ebx),%eax` indicates the variable stored at `ebx` is a pointer, and also variable stored at `edi` in the following snippet is a pointer as it is passed to the first parameter of `strchr` library function. With variable type inference, we could only taint poniter variable of interest when an unknown pointer is dereferenced instead of tainting all variables.

Algorithm 2 Must-not Tainted Analysis

Function *MustNotTainted(UntaintedSet, TaintedInst, ValueSet)*:
    input : set of must-not tainted data object *UntaintedSet*, set of tainted instructions *TaintedInst*, value set *ValueSet*
    output : set of tainted data object *UntaintedSet*, set of tainted instructions *TaintedInst*
14 Source ← TaintSourceScan(Bin)
    Init(buffer_start_addr, buffer_length, ValueSet, Source)
    if *unbounded*($ValueSet^{i}_{entry}$*[buffer_startaddr])* $\bigvee$ *unbounded*($ValueSet^{i}_{entry}$*[buffer_length])* **then**
15   exit()
16 **while** *changed* **do**
17   **foreach** *instruction i* **do**
18   **if** $ValueSet^{i}_{entry}[opaddr] \not\subseteq UntaintedSet$ **then**
19   TaintedInst ← TaintedInst ⊔ {i}
                Transfer(UntaintedSet, ValueSet)
Function *Transfer(UntaintedSet, ValueSet, i)*:
    input : set of must-not tainted data object *UntaintedSet*, value set *ValueSet*
    output : set of tainted data object *UntaintedSet*
20 **switch** *rule(i)* **do**
21   **case** $tag(opaddr_{dest}) \leftarrow tag(opaddr_{dest}) \mid tag(opaddr_{src})$ **do**
22   **case** $tag(opaddr_{dest}) \leftarrow tag(opaddr_{src})$ **do**
23   **case** $tag(opaddr_{unary}) \leftarrow tag(opaddr_{unary})$ **do**
24   UntaintedSet ← UntaintedSet − $ValueSet^{i}_{entry}$[$opaddr_{dest}$]
            if $ValueSet^{i}_{entry}[opaddr_{src}] \subseteq UntaintedSet \bigwedge evalToConcrete(ValueSet^{i}_{entry}[opaddr_{dest}])$ **then**
25   UntaintedSet ← UntaintedSet ⊔ $ValueSet^{i}_{entry}$[$opaddr_{dest}$]
26   **if** $ValueSet^{i}_{entry}[opaddr] = (\top, \top, \top)$ **then**
27   UntaintedSet ← UntaintedSet − $ValueSet^{i}_{entry}$[Overtaint($opaddr_{dest}$)]

```
movl $0xa,0x4(%esp)
mov %edi,(%esp)
call <strchr@plt>
```

Our Algorithm. Specifically, the must-not tainted analysis algorithm as shown in Algorithm 2 first scans the whole binary for possible taint sources, e.g., `read` system call and `recv` system call (line 2). Each identified taint source serves as a starting point of our analysis. The initial tainted buffer has two major characteristics: start address and length. As our evaluation shows, we are able to identify the value set of the introducing tainted buffer starting address and length. Otherwise, if either upper bound or lower bound of buffer start address or buffer length cannot be determined, our analysis triggers a warning and

terminates, since it may indicate program vulnerability (lines 4-5). The analysis is of a work-list style and iterates over each instruction until the `UntaintedSet` and `TaintedInst` remain unchanged (reached a fixed point). For each instruction *i* in the program (line 7), we first compare the incoming value sets of instruction operand address with our must-not untainted value sets, if the former is not a subset of the latter, the instruction is identified as a possible tainted instruction for later taint propagation logic instrumentation (lines 8-9). We then process `UntaintedSet` with respect to the taint propagation rule of each instruction (lines 10-16). Particularly, if the taint propagation rule for instruction *i* decides *i* has a data flow dependence between instruction operand(s), i.e., the taint propagation rule is in the form of:

$$\begin{array}{ll} \text{tag}(opaddr_{dest}) & \leftarrow \text{tag}(opaddr_{dest}) \mid \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{dest}) & \leftarrow \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{unary}) & \leftarrow \text{tag}(opaddr_{unary}) \end{array}$$

we taint destination operand value set and remove it from `UntaintedSet` as shown in lines 13-16. If the source operand is deemed untainted and we know the exact concrete address of destination operand, we enlarge our `UntaintedSet` by adding destination operand value set to `UntaintedSet` as illustrated in lines 17-18. Otherwise, we conservatively taint all of the possible memory address involved in instruction *i*.

Example. We use the instruction `0x8048634: mov %al,-0xd(%ebp)` listed in Table 3.1 to demonstrate how to use our propagation rules to update the must-not tainted set. Specifically, since the source operand of instruction `0x8048634` is not in `UntaintedSet`, this instruction is added to `TaintedInst` for further taint propagation logic instrumentation and since the taint propagation rule for this instructions is in the form of:

$$\text{tag}(op_{dest}) \leftarrow \text{tag}(op_{dest}) \mid \text{tag}(op_{src})$$

According to algorithm 2, `UntaintedSet` is updated:

$$\begin{aligned} \text{UntaintedSet} &= \text{UntaintedSet} - \text{ValueSet}^{i}_{entry}[opaddr_{dest}] \\ &= \text{UntaintedSet} - (\bot, \text{-0x440}, \bot) \end{aligned}$$

3.4.3.2 Soundness Analysis of SELECTIVETAINT

Next, we provide a formal analysis of our must-not tainted analysis to prove that it hardly introduces false negatives, i.e., all instructions in must-not-tainted instruction set $\mathscr{I}_u$ generated by our must-not tainted analysis are indeed not-tainted, except for precision loss due to imprecise CFG and VSA.

Figure 3.4 shows the formal representation of must-not tainted analysis. Basically, for removing or adding an instruction in $\mathscr{I}_u$, one has to apply one of the four primary inference rules, i.e., rule UNREACHABLE, UNKNOWNOPERAND, UNTAINTEDOPERAND, or NONPROPAGATEOPCODE:

- In UNREACHABLE rule, if there is no path from taint sources to instruction i, then instruction i is removed from must-not-tainted instruction set $\mathscr{I}_u$;
- In UNKNOWNOPERAND rule, if there exists an operand with unknown value set, then instruction i is removed from must-not-tainted instruction set $\mathscr{I}_u$;
- In UNTAINTEDOPERAND rule, if for each operand o of instruction i, its value set is a subset of must-not-tainted value set $\mathscr{V}_u$, then instruction i is added to must-not-tainted instruction set $\mathscr{I}_u$;

Primary Inference Rules

Instructions:

$$\textsc{Unreachable}\ \frac{\nexists i_s \in source,\ i_s \rightsquigarrow i}{\mathscr{I}_u \mathrel{-}= \{i\}} \qquad \textsc{UnknownOperand}\ \frac{\exists o \in op(i),\ V[o] = (\bot,\bot,\bot)}{\mathscr{I}_u \mathrel{-}= \{i\}}$$

$$\textsc{UntaintedOperand}\ \frac{\forall o \in op(i),\ V[o] \subseteq \mathscr{V}_u}{\mathscr{I}_u \cup= \{i\}} \qquad \textsc{NonPropagateOpcode}\ \frac{\forall o \in op(i),\ V[o] \overset{i}{=\!=} V[o]}{\mathscr{I}_u \cup= \{i\}}$$

Auxiliary Inference Rules

Control-flows:

$$\textsc{Reachable}\ \frac{succ(i_1,\ i_2)}{i_1 \rightsquigarrow i_2} \qquad \textsc{TransReachable}\ \frac{succ(i_1,\ i_2) \quad succ(i_2,\ i_3)}{i_1 \rightsquigarrow i_3}$$

Operands:

$$\textsc{LiteralOperand}\ \frac{l \in op(i) \quad l : \texttt{literal}}{\mathscr{V}_u \cup= V[l]} \qquad \textsc{LabelOperand}\ \frac{l \in op(i) \quad l : \texttt{label}}{\mathscr{V}_u \cup= V[l]}$$

Opcodes:

$$\textsc{TaintSource}\ \frac{o \in taintedop(i_s) \quad i_s \in source}{\mathscr{V}_u \mathrel{-}= V[o]}$$

$$\textsc{TaintPropagate}\ \frac{o_1 \in sourceop(i) \quad o_2 \in destop(i) \quad V[o_1] \subseteq \mathscr{V}_u}{\mathscr{V}_u \mathrel{-}= V[o_2]}$$

$$\textsc{PCRegChangeOpcode}\ \frac{V[pc] \overset{i}{\neq} V[pc] \quad \forall o \in op(i),\ V[o] \overset{i}{=\!=} V[o]}{\mathscr{I}_u \cup= \{i\}}$$

$$\textsc{StatusRegChangeOpcode}\ \frac{V[status] \overset{i}{\neq} V[status] \quad \forall o \in op(i),\ V[o] \overset{i}{=\!=} V[o]}{\mathscr{I}_u \cup= \{i\}}$$

Notations:

$\mathscr{V}_u$:	Must-not-tainted value set	$\mathscr{I}_u$:	Must-not-tainted instructions
$\overset{i}{=\!=}$:	Equal values after executing i	V:	VSA result map

Figure 3.4: Formal representation of must-not tainted analysis

- In NONPROPAGATEOPCODE rule, if for each operand o of instruction i, there is no side effect on operand o after executing instruction i, then instruction i is added to must-not-tainted instruction set $\mathscr{I}_u$.

The rest inference rules of Figure 3.4 are auxiliary inference rules. Rule REACHABLE indicates, if instruction i_2 is a successor of instruction i_1, then i_2 is reachable from i_1. Similarly, TRANSREACHABLE rule indicates, if instruction i_3 is a successor of instruction i_2 and instruction i_2 is a successor of instruction i_1, i_3 is reachable from i_1. LITERALOPERAND rule indicates, if the operand value set has a type of `literal`, it is added to must-not-tainted value set $\mathscr{V}_u$ and LABELOPERAND rule indicates, if the operand's value set has a type of `label`, it is added to must-not-tainted value set $\mathscr{V}_u$. TAINTSOURCE and TAINTPROPAGATE rules infer that the tainted value set is removed from must-not-tainted value set $\mathscr{V}_u$ and when taint propagates from source operand to destination operand of an instruction, the destination operand is also tainted. PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE rules indicate that, if after executing instruction i, instruction operands value sets are not changed and only the value sets of program counter or status registers are changed, then instruction i is added to must-not-tainted instruction set $\mathscr{I}_u$.

Theorem 1. *Must-not-tainted analysis is sound, w.r.t, precise CFG and VSA results.*

Proof. We prove this theorem with *induction*.

(1) In the first iteration of the analysis, the must-not-tainted set $\mathscr{I}_u$ is $\emptyset$. Must-not-tainted analysis is sound since every instruction is tainted.

(2) We next prove that if in the k-th iteration, the must-not-tainted analysis is sound, w.r.t, precise CFG and VSA results, it also holds in k+1-th iteration.

Suppose the must-not-tainted set $\mathscr{I}_u$ in the k-th and k+1-th iteration are $\mathscr{I}_u^k$ and $\mathscr{I}_u^{k+1}$. Given any instruction i, whether instruction i is added to or removed from must-not-tainted instruction set $\mathscr{I}_u^k$ has four cases:

(2.1) *Instruction i cannot be reached from taint sources.* The UNREACHABLE rule will remove i from $\mathscr{I}_u^k$. In this case, instruction i can be potentially tainted, and therefore safely removing i from $\mathscr{I}_u^k$ will result in a sound $\mathscr{I}_u^{k+1}$.

CFG imprecision. As reconstructing CFG is a hard problem in practice, case 2.1 is sound except for the imprecise CFG. As shown in §3.4.1, SELECTIVETAINT matches callers and callees based on forward-edge CFI identification approaches and matches jumps and jump targets with basic block starting addresses within the same function or function entry addresses. These methods may introduce false negatives and produce imprecise CFG for real-world binaries.

(2.2) *One or more operands of instruction i have an unknown value set.* The UNKNOWN-OPERAND rule will remove i from $\mathscr{I}_u^k$. In this case, instruction i can propagate taints, and therefore safely removing i from $\mathscr{I}_u^k$ will result in a sound $\mathscr{I}_u^{k+1}$.

VSA imprecision. Though VSA may introduce imprecision, this rule conservatively removes all instructions with unknown value sets from $\mathscr{I}_u^k$.

(2.3) *All operands of instruction i are subsets of must-not-tainted value set $\mathscr{V}_u^k$.* $\mathscr{V}_u^k$ is updated based on rules in Operands rule group in Figure 3.4. Per rule LITERALOPERAND and LABELOPERAND, if an operand is of type `literal` or `label`, its value cannot propagate taint and it is added to $\mathscr{V}_u^k$. Per rule TAINTSOURCE and TAINTPROP, at taint source and taint propagation instructions, $\mathscr{V}_u^k$ gets updated by removing the tainted value set. If all operands of instruction i are subsets of must-not-tainted value set $\mathscr{V}_u^k$, it means all values in the operands are must-not-tainted and certainly should be added to $\mathscr{I}_u^k$ and $\mathscr{I}_u^{k+1}$ is sound.

VSA imprecision. As VSA is a undecidable problem, it may introduce imprecision when VSA fails to identify whether an operand is actually a subset of $\mathscr{V}_u^k$ and when $\mathscr{V}_u^k$ is updated.

Algorithm 3 SELECTIVETAINT Algorithm

```
Function SelectiveTaint(Bin):
    input  : original bianry Bin
    output : instrumented binary NewBin
1   Init (UntaintedSet, TaintedInsn, ValueSet)

2   while changed do
3       CFG ← CfgReconstructtion(Bin, CFG, ValueSet)
        ValueSet ← whole_program_VSA(CFG, ValueSet)
        UntaintedSet, TaintedInsn ← MustNotTainted(UntaintedSet, TaintedInst, ValueSet)

4   NewBin ← Rewriting (Bin, TaintedInst)
```

(2.4) *Instruction opcode has no impact on taint propagation.* In this case, instruction i should be added to $\mathscr{I}_u$, as the instruction does not involve in any explicit handling of tainted data. Particularly, an instruction may only have side effects on program counter or status register but not its operands, and in these cases no taint is involved and instruction i should be added to $\mathscr{I}_u^k$, as in rules PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE, which results in a sound $\mathscr{I}_u^{k+1}$.

Therefore, must-not-tainted analysis of SELECTIVETAINT is sound, except for the imprecision introduced by current limitations of undecidable CFG reconstruction and VSA results in binary. □

3.4.4 Binary Rewriting

Based on the identified tainted instructions, we instrument taint propagation logic via binary rewriting for each instruction just as how a conventional DFT performs. The only difference is conventional DFTs instrument at run-time through dynamic binary instrumentation and yet we instrument the binary statically to track how taints are introduced at the taint sources, propagated, and checked at taint sinks.

With the support from our CFG reconstruction, value set analysis, and taint instruction identification, we then sequentially combine these three analyses in a loop body and the set of `ValueSet`, `UntaintedSet` and `TaintInst` get gradually changed until a fixed point is reached. In particular, as illustrated in algorithm 3, at line 2, we first initialize the three sets of `ValueSet`, `UntaintedSet` and `TaintInst` with the taint source information. Lines 3-6 will reach a fixed point after iterations of sequentially applying CFG reconstruction, value set analysis and taint instruction selection algorithm. When all the taint sources are processed, at line 7, our binary rewriter rewrites the original binary with taint propagation logic on selected instructions.

Note that for performance reasons, we use a function summary approach to process standard libraries such as `libc`, which is inspired by how RAMBLR [112] handled library. That is, we will not statically rewrite the instructions in the library, and instead we rewrite the callers to perform direct taint tracking e.g., introduced taint, and propagate taint according to the corresponding parameters. For instance, when we notice `memcpy` call, we will directly taint the destination memory based on the data in the source memory.

3.5 Implementation

We have implemented SELECTIVETAINT atop angr [101] and Dyninst [6]. Specifically, (1) we used angr to build a CFG for the binary, then implemented our own forward-edge CFI identification based on TypeArmor [111] and τCFI [77], i.e., using our own VSA to determine unsolved call sites targets, connecting unsolved call sites to functions with same parameter count and parameter type, and connecting unsolved indirect jumps with basic block starting address or all function entry addresses; (2) we implemented our own flow-sensitive and context-sensitive whole program VSA, which is used to determine the

value set held at each program point; (3) based on the generated CFG and VSA results, we implemented taint instruction identification using the rules described in Figure 3.4 to identify the untainted instructions; (4) and after that we use Dyninst to statically rewrite the binary, which is widely used in recent studies [79, 84, 103, 110, 111]. The total implementation of SELECTIVETAINT consists of 7,000 python code, and 22,000 C/C++ code. The source code of SELECTIVETAINT will be made publicly available.

CFG Construction. To implement the analyzer, first, we recover the control flow graph (CFG) of the binary using angr in which step we find every basic block address, its containing instructions and the predecessors and successors of each basic block. Afterwards, remaining unsolved indirect control flow transfers are further resolved using our method describe in §3.4.1.

Value Set Analysis. We initialize sets `ValueSet` with the data extracted from original binary, e.g., section and segment information, initial data values in `.rodata` and `.data` section. When identifying memory region for variables, (1) for stack variable, we track the value set of stack pointer `SP` in different calling context, examine and figure out whether a variable is a stack variable and in which function the variable is defined, i.e., in whose stack frame the variable resides; (2) for global variable, we track the value set of variables and check if it could be evaluated to an address in code segment or data segment as a global variable; (3) for heap variable, we track the call instructions for `malloc`, `alloc` `C` library calls and etc. to determine whether it is a heap variable.

In intra-procedural analysis, the value set for each abstract location is calculated in a worklist algorithm until a fixed point is reached. In inter-procedural analysis, a function summary is generated based on intra-procedural analysis results to summarize the value

set changes of each function. The static analysis finishes when all value sets in the whole program remain unchanged.

Taint Instruction Selection. We examine maintain a must-not tainted abstract location set for each program point based on the value sets generated by value set analysis and the type of each instruction generated by Capstone disassembler. When must-not tainted abstract location sets reach a fixed-point, each instruction is examined as tainted or untainted based rules in Figure 3.4, i.e., we conservatively assume an abstract location is tainted whenever we cannot determine its taintedness. Unlike libdft which is implemented using Pin, we do not go into the dynamic library functions, and instead, we use a function summary for each library functions to track taint propagation.

Binary Rewriting. Our rewriter is implemented with a bit tag size and a byte tag granularity using Dyninst [6] binary instrumentation and analysis framework. Dyninst is an anywhere, anytime binary instrumentation framework which could be used in both static binary rewriting at compile-time or dynamic instrumentation at run-time. We favor Dyninst as it is a the-state-of-the-art tool in binary rewriting which is used in a variety of tools and its robust API implementation.

3.6 Evaluation

In this section, we present the evaluation results. To see the improvements over dynamic taint analysis, we compare SELECTIVETAINT with `libdft` [54]. The version of Intel Pin used to build `libdft` was 2.14 (build 71313), and we slightly modified the `nullpin` and `libdft` tool and adopted them in our experiment settings. Also, to see the advancements of the selective taint analysis, we also implemented a static taint analysis by instrumenting all instructions, and we call this system STATICTAINTALL. We use the SPEC 2006

Benchmark	Input Functions	# Func.	# Inst.	# SELECTIVETAINT Instrum. Inst. (%)	# libdft Executed Tainted Inst. (%)	Analysis Time (s)
SPEC benchmarks						
400.perlbench	read, fread, fgetc	1,855	229,895	100,373 (43.66%)	3457 (1.51%)	7,085
401.bzip2	read	102	12,492	4,904 (39.26%)	61 (0.49%)	27
403.gcc	read, fread, _IO_getc	4,725	653,275	286,783 (43.90%)	1426 (0.22%)	65,054
429.mcf	fgets	52	2,631	146 (5.55%)	486 (18.49%)	2
445.gobmk	_IO_getc, fgets	2,591	172,105	77,284 (44.91%)	2633 (1.53%)	4,315
456.hmmer	fread, fgets	572	62,098	23,543 (37.91%)	2400 (3.89%)	520
458.sjeng	_IO_getc, fgets	183	24,052	8,374 (34.82%)	65 (0.27%)	196
462.libquantum	fread, fgetc	137	9,760	3,615 (37.04%)	759 (7.78%)	17
464.h264ref	read, fread, __isoc99_fscanf	578	107,486	49,143 (45.72%)	894 (0.83%)	1,511
471.omnetpp	_IO_getc, fgets	2,140	123,635	52,429 (42.41%)	1303 (1.06%)	6,199
473.astar	read, fscanf	121	9,013	2,125 (23.58%)	780 (8.66%)	15
483.xalancbmk	fread, std::istream::read	13,645	704,360	312,113 (44.31%)	1278 (0.18%)	80,962
Web applications						
nginx server	read, pread64, readv, recv	1,276	134,210	55,200 (41.13%)	1,594 (1.19%)	2,468
lynx browser	read, fread, fgetc, fgets, _IO_getc, wgetch, readlink	1,671	217,474	82,726 (38.04%)	2,185 (1.00%)	6,609
Other applications						
SoX 14.4.2	read,fread,fgets,_IO_getc,__isoc99_scanf,__isoc99_fscanf	1,158	113,112	33,529 (29.64%)	1,554 (1.37%)	1,644
TinTin++ 2.01.6	read,fread,fgets,_IO_getc,gnutls_record_recv,fgetc	830	94,161	47,631 (50.58%)	1,268 (1.35%)	1,074
Mini-XML 2.12	read,_IO_getc,__fread_chk	274	20,463	238 (1.16%)	553 (2.70%)	100
dcraw 9.28	fread,fscanf,__fread_chk,_IO_getc,fgets,jpeg_read_header	291	70,882	24,646 (34.77%)	581 (0.82%)	701
ngiflib 0.4	fread,_IO_getc	42	2,149	314 (14.61%)	441 (20.52%)	2
Gravity 0.3.5	read,getline	1,123	87,120	34,307 (39.38%)	1,224 (1.40%)	1,203
ncurses 6.0	read,fread,fgets,fgetc	821	61,825	24,544 (39.70%)	755 (1.22%)	572
LAME 3.99.5	fread,_IO_getc,__fread_chk	553	71,656	20,186 (28.17%)	1,101 (1.54%)	613
MP3Gain 1.5.2	fread,_IO_getc	143	17,609	4,326 (24.57%)	1,286 (7.30%)	44
LibTIFF 4.0.7	read,jpeg_read_header,jpeg_read_raw_data	620	55,674	20,888 (37.52%)	616 (1.11%)	396
NASM 2.14.02	fread,fgets,fgetc	531	81,458	34,203 (41.99%)	765 (0.94%)	908
libjpeg-turbo 2.0.1	fread,_IO_getc,__fread_chk	2,186	118,082	27,640 (23.41%)	503 (0.43%)	981
OpenJPEG 2.3.0	fread,_IO_getc,fgets,fgetc,__isoc99_fscanf,png_read_image	158	18,006	4,322 (24.00%)	772 (4.29%)	51
Jhead 3.00	fread,fgetc	109	9,969	3,412 (34.23%)	471 (4.72%)	19

Table 3.2: Instructions instrumented by SELECTIVETAINT and libdft.

CPUINT benchmarks, network daemon Nginx-1.4.0, web browser Lynx, and 14 recent program vulnerabilities to evaluate SELECTIVETAINT. We first evaluate its effectiveness by looking into the details of how SELECTIVETAINT performs in §4.4.1, and then report the performance overhead of the rewritten binaries in §4.4.3. Finally, we demonstrated its security applications with real world binaries in §3.6.3.

3.6.1 Effectiveness

We report the effectiveness of how SELECTIVETAINT performs with the SPEC2006 CPUINT benchmarks in Table 3.2. The first column shows the 28 C/C++ programs in the benchmark we used in our evaluation, followed by the 2nd column of the input functions detected by SELECTIVETAINT. Note that the input function is the function that introduces

the taint sources. Next, we report the total number of functions contained in the benchmark program in the 3rd column, which provides an estimation of the complexity of the program. Then, we show the total number of instructions identified in the binary in the 4th column. Our STATICTAINTALL statically rewrites all of these instructions, similarly to how dynamic taint analysis instruments them. This will provide an upper bound of how SELECTIVETAINT would perform in the worst case (by statically taint them all). Next, we show the total number of instructions that need to be statically instrumented by SELECTIVETAINT in the 5th column followed by the total number of executed unique instructions that really involved in taint analysis in the 6th column, and this number is obtained by running the corresponding benchmark by using the default configured input with `libdft`, which will provide a lower bound of the number of unique tainted instructions. For fair comparison, we did not count the instructions in the library from the `libdft` trace since SELECTIVETAINT will not instrument them. Finally, we report how long SELECTIVETAINT performs to process each of the benchmarks in the last column.

We can observe from Table 3.2 that our VSA analysis on the must-not tainted instruction analysis works well in these benchmarks, and we have largely reduced the possible tainted instructions to only about 1.66% - 50.58% compared to STATICTAINTALL. While ideally we would like to instrument only the instruction involved in the taint analysis (which is a subset of the instructions identified by SELECTIVETAINT), as detected by the `libdft` which shows about 0.18% - 20.52% of these instructions are essentially needed in the taint analysis at run-time, we will not be able to achieve this by purely static analysis.

False Positives and False Negatives. We define *false negatives* to be the instructions SELECTIVETAINT assumes to not be tainted but `libdft` indicates it should and *false positives* to be the instructions SELECTIVETAINT assumes to be tainted but `libdft` indicates it should

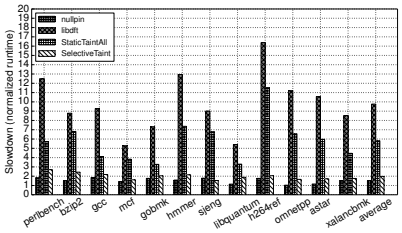
not. By examining the instructions tainted by SELECTIVETAINT and libdft, we observe SELECTIVETAINT reports no false negatives but false positives. False positives indicates SELECTIVETAINT is conservative and over-tainting instructions and false positives are acceptable, while no false negatives indicates our approach is a sound over-approximation of the tainted instructions. We contribute this to the conservative rules in Figure 3.4, for instance, we remove the value set from untainted value set as long as we cannot determine the taintedness of that value set.

Internal Statistics. We also measured the statistics of SELECTIVETAINT in Table 3.3 to understand its inner-workings. Columns 2-3 are CFG reconstruction details, i.e., the number of initial CFG edges and the number of final CFG edges after our CFG reconstruction. We can observe our CFG reconstruction can add hundreds of edges to the CFG using the techniques described in §3.4.1. Columns 4-8 are value set analysis statistics, which are the number of abstract locations in the analysis, the unknown abstract locations due to command line parameters, argument aliasing when missing callers, and library function calls. We can observe our approach identifies thousands of abstract location and multiple unknown abstract locations in each category. Columns 9-12 are numbers in taint instruction identification, such as the number of initially untainted value sets in the first iteration, the number of final untainted value sets, the intra-procedural iteration times, and the inter-procedural iteration times. We can observe that the number of untainted value sets get smaller through analysis iterations, which means our analysis does propagate untaintedness and remove potentially tainted value sets from the must-not-tainted set $\mathscr{V}_u$. The intra-procedural analysis generally has hundreds of iterations while the inter-procedural analysis has fewer (tens of) iterations, from which we can observe the intra-procedural analysis reaches the fixed point with more iterations than inter-procedural analysis.

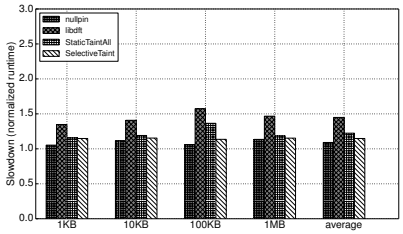
Benchmark	CFG Reconstruction		Value Set Analysis					Taint Instruction Identification			
	Init. Edges	Updated Edges	A-Loc	Uninit. CLI	Uninit. Arg. Alias	Uninit. Lib	Uninit. Total	1st-pass Untaint-V	Last-pass Untaint-V	#Intra. Iter.	#Inter. Iter.
SPEC benchmarks											
400.perlbench	134528	192247	54779	2	2077	1125	3204	34353	17191	9941	4
401.bzip2	4686	5280	1809	3	62	125	190	1469	1079	731	6
403.gcc	446969	757822	167792	2	2957	3221	6180	91355	41822	9450	2
429.mcf	1046	1113	658	2	17	48	67	602	602	110	2
445.gobmk	73068	94056	55641	7	6347	800	7154	45675	20839	19673	6
456.hmmer	28040	32339	14637	4	864	1608	2476	8260	6431	3658	5
458.sjeng	12250	12732	4710	5	67	416	488	4031	1994	1141	5
462.libquantum	3481	3638	2357	4	158	115	277	2237	1162	981	6
464.h264ref	33980	64541	16970	3	116	629	748	9667	7062	3888	5
471.omnetpp	76573	560618	51777	2	1532	5144	6678	21562	12754	14066	5
473.astar	3156	3374	2266	5	94	187	286	1913	1566	853	6
483.xalancbmk	356198	6616514	335098	2	2260	41491	43753	130351	104012	27290	2
Web applications											
nginx server	59593	233072	31304	2	806	1061	1869	19105	13884	8364	5
lynx browser	140063	438081	59413	5	287	6269	6561	20448	11092	9440	4
Other applications											
SoX 14.4.2	47050	68338	25654	6	1474	1828	3308	16512	13074	9544	7
TinTin++ 2.01.6	34351	34407	17712	3	465	2203	2491	8674	1135	4600	4
Mini-XML 2.12	11401	18911	3684	4	7	597	608	3616	3542	706	3
dcraw 9.28	24830	26423	9414	2	111	1506	1619	4496	4197	1516	4
ngiflib 0.4	1022	1038	453	4	17	94	115	276	276	97	2
Gravity 0.3.5	36380	53161	22184	5	1415	537	1957	10805	6276	4645	4
ncurses 6.0	32129	58793	15613	6	141	630	777	10759	6667	5996	6
LAME 3.99.5	26449	31662	11060	3	333	677	1013	8087	5837	3374	5
MP3Gain 1.5.2	6293	6522	2948	4	19	514	537	1841	1595	838	5
LibTIFF 4.0.7	23939	78337	12471	4	549	496	1049	6707	5237	3241	4
NASM 2.14.02	43112	97209	17354	5	154	935	1094	8607	3833	2848	4
libjpeg-turbo 2.0.1	34083	280515	15908	5	1319	613	1937	14455	9893	5007	5
OpenJPEG 2.3.0	7461	7719	3178	5	214	964	1183	1496	1271	686	4
Jhead 3.00	5292	5571	2383	3	17	467	487	1613	873	529	4

Table 3.3: SELECTIVETAINT Internal statistics.

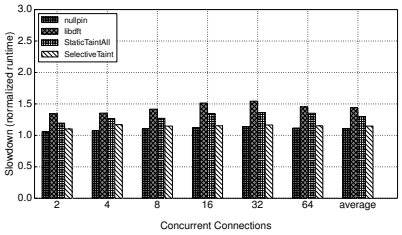
Performance. With respect to the performance (e.g., the analysis time) of SELECTIVETAINT itself, we notice it is a very time consuming process, especially with large software. This is understandable, since SELECTIVETAINT will inspect each instruction and calcaulate VSA for each of them. Meanwhile, the analysis has to be run twice: first calculating the VSA, and then determining the taintedness. We notice it took more than 5 hours to process the `403.gcc` benchmark, whereas for small binaries, e.g., `462.libquantum`, it could take just a few minutes.



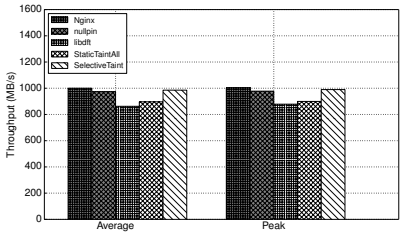
(a) The slowdown imposed by libdft and SELECTIVETAINT when running SPEC CPU 2006 benchmark suite



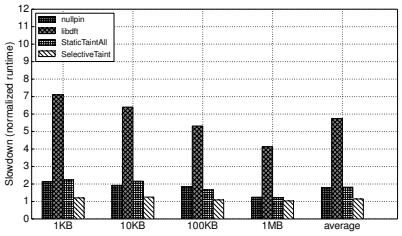
(b) The slowdown imposed by libdft and SELECTIVETAINT when running Nginx with different requested file sizes



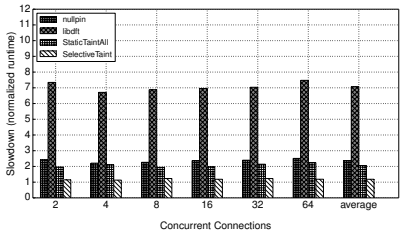
(c) The Nginx slowdown imposed by libdft and SELECTIVETAINT with varying concurrent connections



(d) Throughput performance of Nginx when downloading HTTP files



(e) The Lynx slowdown imposed by libdft and SELECTIVETAINT when requesting different sizes of web pages



(f) The slowdown imposed by libdft and SELECTIVETAINT with varying concurrent connections

Figure 3.5: Performance Overhead Evaluation of the Tested Benchmarks.

3.6.2 Efficiency

Next, we measure the performance overhead of the rewritten binaries. To compare with `libdft`, we run the binaries with the default configured input, with nullpin (a simple implementation to evaluate Intel Pin platform overhead), `libdft` with a bit level taint. We run the corresponding benchmark with and without rewriting to understand the additional

overhead. All of the experimental results were obtained with 10 runs and then normalized by dividing each average result against native unmodified executables.

SPEC2006 CPUINT benchmarks. Figure 3.5a shows the normalized runtime overhead of nullpin, `libdft`, STATICTAINTALL, SELECTIVETAINT and SELECTIVETAINT-TYPED, when running with the SPEC2006 CPUINT benchmark suite, compared with the native execution. We can notice that `libdft` imposes slowdown of ranging from 5.26x (`mcf`) to 16.37x (`h264ref`), whereas STATICTAINTALL and SELECTIVETAINT impose 3.27x (`gobmk`) to 11.53x (`h264ref`) and 1.54x (`sjeng`) to 2.70x (`perlbench`), respectively. STATICTAINTALL outperforms `libdft` in all 12 benchmarks with 1.29x - 2.25x faster and similarly SELECTIVETAINT performs even 2.88x - 7.98x faster than that of `libdft`.

Nginx web server. One ideal use case for SELECTIVETAINT would be for the protection of network daemons. We thus use Nginx as a benchmark to thoroughly evaluate its overhead. In particular, we tested nullpin, libdft, STATICTAINTALL, and SELECTIVETAINT on Nginx web server 1.4.0 with default settings. When requesting different file sizes with 1KB, 10KB, 100KB, and 1MB, respectively, using Apache Benchmark `ab`, the result is illustrated in Figure 3.5b. All four tools including libdft performs no more than 2x slowdown. The biggest slowdown for libdft is 1.56x (100KB). STATICTAINTALL imposes 1.16x-1.37x slowdown and outperforms libdft by 1.15x-1.24x. SELECTIVETAINT performs even better with 1.14x-1.15x which outperforms libdft by 1.17x-1.39x.

We tested the concurrent performance of nullpin, libdft,STATICTAINTALL, and SELECTIVETAINT with 2, 4, 8, 16, 32 and 64 concurrent connections. The result is shown in Figure 3.5c. The nullpin performs 1.16x-1.14x slowdown. While libdft imposes 1.34x-1.55x slowdown, STATICTAINTALL imposes a less slowdown of 1.2x-1.36x. SELECTIVETAINT outperforms nullpin, libdft and STATICTAINTALL by a slowdown of 1.10x-1.17x.

We also tested the throughput of Nginx downloading files over HTTP connections as shown in Figure 3.5d. The worst average throughput performance is libdft which is 13.8% smaller requests rate (862.3 req/s). SELECTIVETAINT has 1.5% smaller requests rate (985.4 req/s). The nullpin and STATICTAINTALL have 2.7% (973.4 req/s) and 10.4% (896.4 req/s) smaller requests rate, respectively.

Lynx web browser. To test the performance of SELECTIVETAINT on client side applications, we also tested Lynx web browser. Specifically, we tested nullpin, libdft, STATICTAINTALL, and SELECTIVETAINT on Lynx web browser 2.8.8 with default settings. When requesting different file sizes with 1KB, 10KB, 100KB, and 1MB, respectively, the result is illustrated in Figure 3.5e. libdft performs biggest slowdown of 4.13x-7.10x, while nullpin and STATICTAINTALL impose 1.25x-2.14x and 1.21x-2.24x slowdown, which outperform libdft by 2.88x-3.33x and 2.96x-3.41x, respectively. SELECTIVETAINT has an even better slowdown of 1.03x-1.25x which outperforms libdft by 4x-5.9x.

We tested the concurrent performance of nullpin, libdft, STATICTAINTALL, and SELECTIVETAINT with 2, 4, 8, 16, 32 and 64 concurrent Lynx web browser instances. The result is shown in Figure 3.5f. The nullpin performs 2.21x-2.51x slowdown. While libdft imposes 6.71x-7.48x slowdown, STATICTAINTALL imposes a less slowdown of 1.94x-2.23x. SELECTIVETAINT outperforms nullpin, libdft and STATICTAINTALL by a slowdown of 1.13x-1.23x.

3.6.3 Security Case Studies

Protecting Nginx web server. To show that our tools could be used to detect real-world attacks, we first implemented a buffer overflow attack detector and used it to protect `Nginx` web server. To test its effectiveness, we generated an exploit based on the buffer overflow

Program	Vulnerability	CVE ID
SoX 14.4.2	Buffer Overflow	CVE-2019-8356
TinTin++ 2.01.6	Buffer Overflow	CVE-2019-7629
Mini-XML 2.12	Buffer Overflow	CVE-2018-20593
dcraw 9.28	Buffer Overflow	CVE-2018-19655
ngiflib 0.4	Buffer Overflow	CVE-2018-11575
Gravity 0.3.5	Buffer Overflow	CVE-2017-1000437
ncurses 6.0	Buffer Overflow	CVE-2017-16879
LAME 3.99.5	Buffer Overflow	CVE-2017-15046
MP3Gain 1.5.2	Buffer Overflow	CVE-2017-14411
LibTIFF 4.0.7	Buffer Overflow	CVE-2016-10095
NASM 2.14.02	Use-After-Free	CVE-2019-8343
libjpeg-turbo 2.0.1	Integer Overflow	CVE-2018-20330
OpenJPEG 2.3.0	Integer Overflow	CVE-2018-5785
Jhead 3.00	Integer Underflow	CVE-2018-6612

Table 3.4: Tested vulnerable software and their vulnerabilities.

vulnerability CVE-2013-2028. By leveraging this vulnerability, an attacker could send a malformed request that triggers an integer signedness error which further causes a stack-based buffer overflow. This bug can be used in a denial-of-service attack or cause arbitrary code execution. Without any surprise, our SELECTIVETAINT detects the exploit at the `ret` instruction because the return value stored on the stack is tainted and thus triggers a warning.

Protecting other binaries against recent memory exploits. We further tested 14 recent real world software vulnerabilities from Common Vulnerabilities and Exposures (CVE)[1], which are listed in Table 3.4. The collected vulnerabilities covered a broad range of software vulnerabilities, including buffer overflow vulnerability, use-after-free vulnerability, integer overflow vulnerability and integer underflow vulnerability, which manifested in varied programs such as sound processing utilities `SoX`, programming language interpreter `Gravity`, and text-based user interfaces `ncurses`.

[1] https://cve.mitre.org/

We implemented the corresponding exploits to compromise these vulnerabilities and validate whether SELECTIVETAINT is able to detect the attacks. For instance, to exploit `CVE-2018-20593` vulnerability in `Mini-XML 2.12`, we developed a malformed `xml` file, which overflowed program stack to rewrite the return address with payloads in `xml` file. To exploit `CVE-2019-7629` vulnerability in `TinTin++ 2.10.6`, we set up a simple game server with exploits that keep sending crafted message which overflowed the multiplayer online game client `TinTin++`. Then test binaries were instrumented with SELECTIVETAINT. In all cases, SELECTIVETAINT successfully detects the exploits which shows SELECTIVETAINT can facilitate real world taint analysis in various applications.

3.7 Summary

We have presented an efficient static analysis based data flow tracking framework SELECTIVETAINT. Unlike previous taint analysis that uses dynamic binary instrumentation, SELECTIVETAINT is built atop static binary rewriting. The key enabling technique is the use of VSA to identify the instructions that never involve taint analysis, and then rewrite the rest to implement the taint analysis. We have tested SELECTIVETAINT with 28 binary programs including SPEC2006 CPUINT benchmarks and observed a substantial performance gain, which is 5X faster than state of the art dynamic taint analysis.

Chapter 4: Exploring Value Set Analysis for Data Race Detection in Intel SGX Enclave Binary

4.1 Changllenges and Insights

In order to detect a data race between threads, intuitively we have to first identify the shared variables, then exclude the ones that are lock protected, and finally interleave the threads to detect whether there is the access that can cause a data race. Therefore, at a high level there will be three challenges when identifying controlled data races in enclave binaries. In the following, we use a working example presented in Figure 4.1, which is a very simple SGX program with two ecalls, to clearly illustrate these three challenges.

Identifying shared variables in enclave binaries. Unlike variables in program source code which are declared and very obvious to be identified (e.g., variable `global_mutex_0` used at line 2 in Figure 4.1), the variables in the binaries are hard to be identified, since the symbols are all gone and variables have all been translated into just registers and memory addresses. We have to rely on the access of memory addresses and registers to abstract the variables, based on each instruction semantics. However, this has been proved to be a challenging task. Also, to detect a data race, we have to know the specific accesses (e.g., R-Read, or W-Write) to the variables, since only R/W or W/W accesses will cause races.

```
1  void ecall_0(void) {
2      sgx_thread_mutex_lock(&global_mutex_0);
3      global_counter = 0;
4      ...
5      if(global_counter == 0) foo();
6      sgx_thread_mutex_unlock(&global_mutex_0);
7  }
8  void ecall_1(void) {
9      sgx_thread_mutex_lock(&global_mutex_1);
10     global_counter = 1;
11     sgx_thread_mutex_unlock(&global_mutex_1);
12 }
```

Figure 4.1: Working example.

To identify variables in the binaries is not a new problem. One promising technique to identify them statically in binary is through value set analysis (VSA) [3]. Recently, there have been significant development with VSA in binary analysis (e.g., [37, 112, 124]), and certainly we can leverage these advances to identify shared variables when developing SGX-RACER. To further differentiate the specific access of the variables, we can rely on the instruction semantics, such as "`inc %rax`" implying both a read and write access of `rax`, "`mov $1, %rax`", and "`pop %rax`" implying write accesses of `rax`, and "`cmp $0, %rax`" implying a read access of `rax`.

Identifying lock variables in the enclave binaries. After identifying the shared variables, we must further identify whether they are protected by any synchronization primitives, and then exclude them from our analysis. However, there are many synchronization primitives, such as thread once, barriers, spin locks, mutex locks, reentrant mutex locks, read-write locks, and condition variables. These primitives are typically provided by SGX SDKs. Additionally, there are also programmer defined synchronization primitives such as locks built from `LOCK` prefix (e.g., an instruction sequence "`mov $0x1,%ecx`", "`lock cmpxchg`

%ecx,(%rdx)" which moves a constant value 1 to the lock variable indexed by register rdx and then locks it), and also the special xchg instruction which asserts the lock signal regardless of LOCK prefix.

For standard synchronization primitives, fortunately SGX SDK provides the corresponding APIs. For instance, as shown in our working example Figure 4.1, there are APIs such as sgx_thread_mutex_lock, and sgx_thread_mutex_un lock. We can easily identify them based on the provided APIs. To identify self-defined synchronization primitives, we can first identify the instructions with LOCK prefix, and perform data flow analysis to identify the lock variables associated with the lock instructions and the specific lock/unlock values written to them (e.g., 0 to unlock, and 1 to lock). After identifying the involved synchronization primitives, we can then use a lockset-based algorithm [51] to exclude the shared variable accesses that cannot be raced.

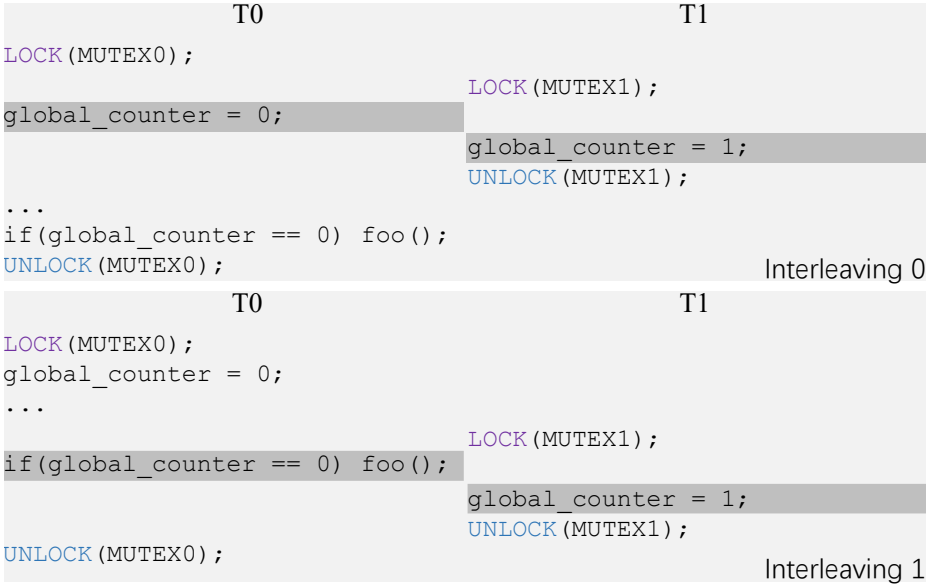
Statically detecting data races due to unintended thread interleavings. As ecalls can be invoked in an arbitrarily order by a malicious OS, all ecalls become concurrent, which introduces many more possible thread interleavings than a traditional data race detector needs to consider. Intuitively, for an SGX program with n ecalls ($n > 1$), there are $C_n^2 + C_n^1 = \frac{n(n+1)}{2}$[2] possible concurrent ecall combinations, some of which are not originally intended to be concurrent and now become unintended interleavings. For instance, Figure 4.1 is a simple enclave program which has two ecalls: ecall_0 and ecall_1. Therefore there are three concurrent ecall combinations: concurrent ecall_0 and ecall_0, concurrent ecall_0 and ecall_1, and concurrent ecall_1 and ecall_1. In the original program, ecall_0 and ecall_1 are never assumed to be concurrent and should only yield the intended interleaving

[2]Note that C_n^2 denotes the total number combinations of any two independent ecalls (e.g., <ecall_0, ecall_1>), and we have to add additional C_n^1 ecall pairs which are those the same ecalls (e.g., <ecall_0, ecall_0>), to get the total number of all combinations.

```
LOCK(MUTEX0);
global_counter = 0;
...
if(global_counter == 0) foo();
UNLOCK(MUTEX0);

LOCK(MUTEX1);
global_counter = 1;
UNLOCK(MUTEX1);
```

(a) No data race in the intended single-threaded program (`ecall_0` first and then `ecall_1`)



(b) Data races in the unintended multi-threaded program (T0 invokes `ecall_0` and T1 invokes `ecall_1`)

Figure 4.2: Thread interleavings in working example.

in Figure 4.2a where no data races occur on shared variable `global_counter`. However, in controlled data race attacks, the attacker can concurrently invoke `ecall_0` and `ecall_1` at very fine-grained granularity, and can result in the unintended interleavings in Figure 4.2b which have data races on shared variable `global_counter`.

While we can use a brute force approach to consider all possible ecall combinations and interleavings, one optimization we can have is to only consider ecalls that access at least

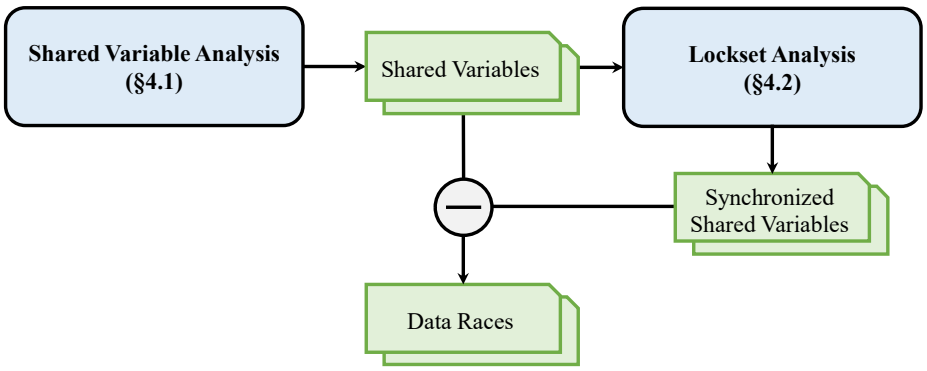


Figure 4.3: Overview of SGX-RACER.

one common shared variable and at least one access is a write to filter out concurrent ecalls that will never introduce data races from all possible combinations. That is, we can focus on the combinations of the shared variable access and explore their possible combinations, instead of intuitively using ecall combinations. This helps us efficiently prune concurrent ecall combinations that will never cause data races.

4.2 System Overview

An overview of SGX-RACER is illustrated in Figure 4.3. To detect a data race in enclave binaries, it consists of two phases of analysis: (*i*) shared variable analysis to identify the set of all of the shared variables and their accesses, and (*ii*) lockset analysis to identify the set of all of the synchronization protected shared variables and their accesses. The intersection of these two sets will be the final detected data races.

Shared variable analysis. To identify data races, SGX-RACER first needs to detect the shared variables that can potentially be accessed in data races. Shared variables in a program typically belong to either global variables or heap variables, and they are identified by analyzing memory accesses in the binary via data flow analysis and the corresponding

Variable Name	Line#	R/W
global_counter	3	W
global_counter	5	R
global_counter	10	W
global_mutex_0	2	R
global_mutex_0	6	R
global_mutex_1	9	R
global_mutex_1	11	R

Table 4.1: Shared variable accesses in working example.

instruction semantics are also used to determine whether it is a read access, or write access, or both. Since fundamentally a data race occurs at the shared variable accesses, not the variable itself, we have to identify their accesses. To this end, we use a set to capture the shared variable accesses, and each element of this set consists of a 3-tuple record: `(shared_variable_address, shared_variable_access_instruction_address, access_type)`, where the shared variable address is either the static memory address in the binary for global variables or the instruction address of the allocation site for heap variables. The access type is either *R*, *W*, or *RW*, representing whether the shared variable is read, written, or both.

As listed in Table 4.1 for our working example, there are seven shared variable accesses, each of which has a variable name, the instruction line number (we use source code line for the illustration purpose and they should be the instruction address in real binary), and the read/write type of that access, e.g., there are three accesses on shared variable `global_counter`, from line 3, 5, 10, respectively.

Lockset analysis. After identifying shared variables from binary, we need to further identify the variables that are synchronization-protected and exclude them from the data race detection. To this end, we need to identify the synchronization primitives (typically locks) held at each shared variable access, by first identifying the lock variables, through analyzing

either synchronization function parameters (e.g., `global_mutex_0` which is identified as a lock variable since it is passed to known synchronization function `sgx_thread_mutex_lock`), or through data flow analysis of instruction sequences that forms programmer defined locks (e.g., instruction sequences with certain `xchg` instructions). After that, SGX-RACER generates a lockset and a lock acquisition history for each instruction. A lockset is a set of possible locks held at each instruction, and a lock acquisition history is a set of locks acquired (and also possibly released) after the last acquisition of a particular lock. With these lockset and lock acquisition history sets, SGX-RACER generates the final set of synchronization protected shared variables, and excludes them from the shared variable access set, to produce the final detected data races.

4.3 Design

4.3.1 Shared Variable Analysis

To find out data races on shared variables, we first need to detect all shared variable accesses in the enclave binary, since the data race must come from these shared variable accesses. Variables in a program can be divided into three categories: global variables, heap variables, and stack variables. Since stack variables are local, they typically will not be shared and we just have to analyze global variables and heap variables. To identify (or index) a global variable, we can use its static memory address, but for heap variables, their addresses are dynamic and we cannot use their run-time addresses and instead we can use their allocation site to identify (index) them. Also, since a data race occurs when particular instructions access an unprotected shared variable, we have to identify the shared variables at each instruction. A data flow analysis is thus needed to identify the definitions (i.e., the *W* accesses) and uses (i.e., the *R* accesses) of these shared variables among the instructions.

Algorithm 4 Shared variable analysis

```
Function SharedVariableAnalysis(binary):
    /* Identify shared variable accesses via data flow analysis                    */
5   foreach instruction i ∈ binary do
6       case i defines v do
            // Update variable value
7           Value(v) ← Update(Value(v))
                // Update shared variables
8           if DestOp(i) ∈ SharedVariable ∧ SrcOp(i) ∈ HeapVariable then
9               SharedVariable ← SharedVariable ⊔ SrcOp(i)
            // If defines a global variable
10          if MemoryAccess(i) ∧ Addr(v) ∈ [global_start, global_end] then
11              SharedVariable ← SharedVariable ⊔ v
                  case R ∉ MemoryAccess(i) ∧ W ∈ MemoryAccess(i) do
12                    SharedAccess ← SharedAccess ⊔ (v, i, W)
                  case R ∈ MemoryAccess(i) ∧ W ∈ MemoryAccess(i) do
13                    SharedAccess ← SharedAccess ⊔ (v, i, RW)
            // If defines a heap variable
14          if CallInst(i) ∧ CallTarget(i) = malloc then
15              HeapVariable ← HeapVariable ⊔ v
16          if v ∈ HeapVariable ∧ v ∈ SharedVariable then
17              case R ∉ MemoryAccess(i) ∧ W ∈ MemoryAccess(i) do
18                  SharedAccess ← SharedAccess ⊔ (v, i, W)
19              case R ∈ MemoryAccess(i) ∧ W ∈ MemoryAccess(i) do
20                  SharedAccess ← SharedAccess ⊔ (v, i, RW)
21      case i uses v do
            // If uses a global variable
22          if MemoryAccess(i) ∧ Addr(v) ∈ [global_start, global_end] then
23              SharedVariable ← SharedVariable ⊔ v
                  case R ∈ MemoryAccess(i) do
24                    SharedAccess ← SharedAccess ⊔ (v, i, R)
            // If uses a heap variable
25          if v ∈ HeapVariable ∧ v ∈ SharedVariable then
26              case R ∈ MemoryAccess(i) do
27                  SharedAccess ← SharedAccess ⊔ (v, i, R)
    /* Generate shared variable access pairs                                       */
28  foreach (v, i0, acc0) ∈ SharedAccess ∧ (v, i1, acc1) ∈ SharedAccess do
29      AccessPair ← AccessPair ⊔ ((v, i0, acc0), (v, i1, acc1))
```

Therefore, we have designed a data flow analysis based algorithm 4 to identify the shared variable accesses. At a high level, the algorithm inspects every instruction in enclave binary and finds variable definitions and variable uses (line 5-28). At each data definition site, the data flow analysis first updates the variable values according to the corresponding instruction semantics such as data arithmetic using VSA [3] (line 7). Then at each data definition (line 8-21) and use site (line 22-28), the algorithm further identifies whether the

data use of the variable is a global variable or a heap variable, whether the variable is a shared variable, and whether the access is an *R*, a *W*, or both. The output of the data flow analysis is a set of shared variable accesses, as what have been shown in Table 4.1 for our working example. Next, we present in greater detail how the algorithm identifies global and heap variables.

Global variable identification. A global variable is identified at the variable definition site if the data flow analysis shows the variable address value could be resolved to a value in binary data sections (data pointers in, e.g., `.text`) or text sections (code pointers in, e.g., `.data`, `.bss` and `.rodata`), as shown in line 10-11. Global variables are accessible from different threads and thus are shared variables on their own. Note that a global variable can be accessed via direct memory access (e.g., "`mov (0x248bb0),%rax`" where `0x248bb0` is a global memory address, or indirect access via an instruction sequence (e.g., "`lea $0x248fa0,%rax`" and "`mov (%rax),%r8`") where a global address `0x248fa0` is first loaded into `rax`, and then dererferenced to load its value into `r8`. We follow the standard data flow analysis to identify them.

Having recognized the global variables from the binary, SGX-RACER further needs to distinguish the different types of the access (i.e., *R*, *W*, or *RW*) based on each specific instruction semantics. In particular, at the data definition sites, the data flow analysis checks whether the variable is written, or both read and written, and if so it assigns the variable access type with *W* or *RW*, as shown in line 11-14; whereas at the data use sites, it only needs to check whether it is an *R* access (line 24-25).

Heap variable identification. Heap variables are not necessarily shared across threads, and only when they are passed through pointer references as function parameters or to a global variable. As such, SGX-RACER tracks heap variables during the data flow analysis to check

whether the pointer of a heap variable is passed to a shared variable (e.g., global variable or function parameter) at the variable definition sites. If so, the heap variable becomes a shared heap variable (described in line 8-9). For instance, in the following assembly code of function `do_save_tcs`:

```
88d4 <_ZL11do_save_tcsPv>:
...
896a: callq 100b1 <dlmalloc>
896f: mov %rax,-0x10(%rbp)
...
89a5: mov -0x10(%rbp),%rax
89a9: mov %rax,(0x248b48)#<_ZL10g_tcs_node>
```

a heap variable is allocated at instruction address `896a` and not yet a shared variable. Until at address `89a9`, when the heap variable pointer is assigned to a global variable at `0x248b48`, this heap variable becomes a shared variable and any afterwards dereferences of the heap variable pointer via the global variable at `0x248b48` are shared across threads. Also, similar to global variables, the data flow analysis further infers the heap variable access type (i.e., *R*, *W*, or *RW*) from the instruction semantics (at line 17-21, and line 26-28).

Generating shared variable access pairs. Based on the data flow analysis results, our shared variable analysis generates possible data race pairs (line 29-line 30), namely *shared variable access pairs*, which is a set of access pairs between two threads (i.e., thread_0 access and thread_1 access, assuming two threads) on the same shared variable. In particular, if a shared variable has n accesses, there are totally $\binom{n}{2} + \binom{n}{1}$ shared variable access pairs for this variable, i.e., a combination of different shared variable accesses $\binom{n}{2}$ plus a combination of identical shared variable accesses $\binom{n}{1}$.

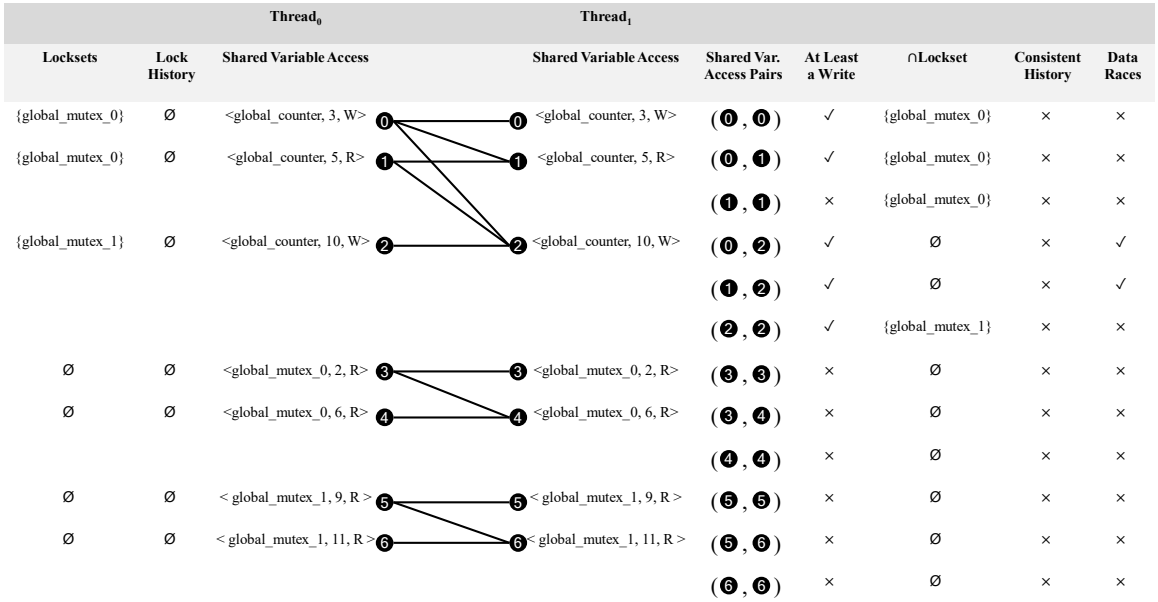


Figure 4.4: The step-by-step internal results showing how SGX-RACER detects the two data races for our working example

For instance, in our working example, the data flow analysis generates in total seven shared variable accesses, as shown in Table 4.1, where `global_counter` has three accesses, `global_mutex_0` has two accesses, and `global_mutex_1` has two accesses. Then, shared variable analysis further generates shared variable access pairs for them: `global_counter` has $\binom{3}{2} + \binom{3}{1} = 6$ entries, `global_mutex_0` has $\binom{2}{2} + \binom{2}{1} = 3$ entries, and `global_mutex_1` has $\binom{2}{2} + \binom{2}{1} = 3$ entries, and thus there are $6+3+3 = 12$ total entries, as illustrated in Figure 4.4. We denote the seven shared variable accesses as nodes, and the 12 edges between nodes shows the combinations of shared variable access pairs, which is also listed in the sixth column.

Algorithm 5 Lockset analysis

Function *LockAnalysis(binary, AccessPair)*:

```
    /* Generate locksets and lock acquisition history via data flow analysis                    */
31  foreach instruction i ∈ binary do
32      if i defines v then
            // Update variable value
33          Value(v) ← Update(Value(v))
34      if (CallInst(i) ∧ CallTarget(i) = lock) ∨ SelfDefineLock(i) then
            // GEN lock variables
35          LockVariable ← LockVariable ⊔ Op(i)
                // GEN locksets
36          LockSet(i) ← LockSet(i) ⊔ Op(i)
              foreach lock l do
37              if l = Op(i) then
                    // KILL lock acquisition history
38                  LockHistory(i,l) ← ∅
39              else
                    // GEN lock acquisition history
40                  LockHistory(i,l) ← LockHistory(i,l) ⊔ Op(i)
41      if (CallInst(i) ∧ CallTarget(i) = unlock) ∨ SelfDefineUnlock(i) then
            // KILL lock variables
42          LockVariable ← LockVariable − Op(i)
                // KILL locksets
43          LockSet(i) ← LockSet(i) − Op(i)
    /* Generate synchronized shared variable access pairs                                       */
44  foreach ((v, l0, acc0), (v, l1, acc1)) ∈ AccessPair do
        // check if at least one access is a write
45      if acc0 = R ∧ acc1 = R then
46          SynAccessPair ← SynAccessPair ⊔ ((v, l0, acc0), (v, l1, acc1))
        // check if there are common locks
47      else if LockSet(i) ⊓ LockSet(i) ≠ ∅ then
48          SynAccessPair ← SynAccessPair ⊔ ((v, l0, acc0), (v, l1, acc1))
        // check if lock acquistion histories are consistent
49      foreach different locks la, lb do
50          if la ∈ LockHisotry(i0, lb) ∧ lb ∈ LockHisotry(i1, la) then
51              SynAccessPair ← SynAccessPair ⊔ ((v, l0, acc0), (v, l1, acc1))
    /* Generate data races                                                                      */
52  DataRace ← AccessPair − SynAccessPair
```

4.3.2 Lockset Analysis

Having generated the shared variable access pairs from variable analysis, SGX-RACER needs to further find out whether any of them have been protected by synchronization primitives (essentially locks), and if so remove them from the detection results. To this end, we have to first identify whether there is any lock associated with the shared variable accesses. A lock usually has a lock variable. Therefore, essentially we have to perform

```
1  /* Enclave.cpp: */
2  void ecall_0(void) {
3      sgx_thread_mutex_lock(&global_mutex_0);
4      sgx_thread_mutex_lock(&global_mutex_1);
5      ...
6      sgx_thread_mutex_unlock(&global_mutex_1);
7      global_counter = 0;
8      sgx_thread_mutex_unlock(&global_mutex_0);
9  }
10 void ecall_1(void) {
11     sgx_thread_mutex_lock(&global_mutex_1);
12     sgx_thread_mutex_lock(&global_mutex_0);
13     ...
14     sgx_thread_mutex_unlock(&global_mutex_0);
15     global_counter = 1;
16     sgx_thread_mutex_unlock(&global_mutex_1);
17 }
```

Figure 4.5: Motivating example of lock acquisition history

liveness analysis of lock variable (§4.3.2.1). If a lock variable is live at the shared variable access, it means this access is lock protected. With the livenesses of the lock variables, we then compute the corresponding lockset for the shared variable access. If the intersection of the lockset of the interleaved shared variable accesses is empty, it means these accesses are not properly synchronized (leading to data races). As such, we use a lockset analysis algorithm (§4.3.2.2) to detect the final data race.

4.3.2.1 Liveness Analysis of Lock Variables

There are two sets of lock variables: one is defined by the standardized synchronization APIs, and the other is defined by programmers. We therefore use the following two strategies to identify them.

Identifying lock variables defined by synchronization APIs. SGX SDKs typically provide a variety of synchronization primitives (as shown in Table 2.1 in **??**). Fundamentally, all of these synchronization primitives can be translated into a lock representation. Therefore, to have a uniformed algorithm, we have to first translate and map them into locks. Specifically, 1) thread-once is mapped as the call-once function holding a unique mutex lock; 2) barrier is mapped as holding *N* mutex locks and *N* is the number of predefined waiting threads; 3) spinlock is mapped as a mutex lock on the `spinlock` object; 4) reentrant mutex is also mapped as a mutex lock; 5) read-write lock is mapped as a read or write lock depending on whether it is used for read or write; 6) conditional variable is mapped as `lock` and `unlock` operations on the associated conditional variable.

Identifying self-defined lock variables. Besides off-the-shelf synchronization primitives, SGX program can also use other self-defined synchronization primitives, e.g., locks built from `xchg` instruction. To identify them, SGX-RACER scans the binary for `xchg`, `lock xchg`, `cmpxchg` and `lock cmpxchg` instructions and considers these instructions to be a lock synchronization primitive through data flow analysis. A lock or unlock is identified according to the associated instruction operand. The lock or unlock semantics is based on the value of the operand (e.g., 1 means lock, and 0 means unlock).

Liveness Analysis of Lock Variable. With the identified lock variables, we perform a liveness analysis with them. The detailed algorithm is presented in algorithm 5. More specifically, at each variable definition site, if a lock variable is defined by either standard APIs or user-defined (line 34), then we generate a GEN set (line 35). This variable will be live for the remaining instructions until it is killed by either unlock APIs or user-defined unlocks (line 41), and we correspondingly generate a KILL set (line 42). For instance, in our working example, line 2 and line 9 are call sites of `lock` synchronization function

	Intel SGX SDK	Open Enclave SDK	Rust-SGX SDK	Rust EDP SDK
Variables				
# Shared Var. Access (R)	317	591	362	161
# Shared Var. Access (W)	119	214	134	21
# Shared Var. Access (R&W)	6	7	16	7
# Uniq. Shared Var.	143	197	138	81
# Lock Var. Access (Mutex)	7	9	0	20
# Lock Var. Access (Spinlock)	53	105	19	0
# Lock Var. Access (Others)	0	4	1	5
# Uniq. Lock Var.	9	14	6	1
Lockset and Acquisition History				
Ins. Lockset Size (Max.)	2	7	5	1
Ins. Lockset Size (Min.)	0	0	0	0
Ins. Lockset Size (Ave.)	0.46	0.36	0.93	0.30
Acqui. History Size (Max.)	8	13	5	0
Acqui. History Size (Min.)	0	0	0	0
Acqui. History Size (Ave.)	3.34	0.1	1.47	0.00
Performance (effectiveness and efficiency)				
# Detected Data Race Shared Variables	18	32	12	2
# Detected Data Races	39	134	27	6
Variable Analysis Time (m)	508.8	12614.4	453.6	441.2
Data Race Detection Time (m)	0.4	2.2	0.2	0.2
Total Time (m)	509.2	12616.6	453.8	441.4

	Intel SGX SDK	Open Enclave SDK	Rust-SGX SDK	Rust EDP SDK
Variable Distribution				
Data Variable	libsgx_trts.a (4)	liboecore.a (17)	libenclave.a (1)	std::sys (1)
	libtlibc.a (4)	liboeenclave.a (3)	libsgx_trts.a (4)	std::panicking (1)
	libunwind.a (2)	liboelibc.a (2)	libtlibc.a (4)	std::thread (0)
	libcpprt.a (1)	liboesyscall.a (1)	libunwind.a (1)	std::sys_common (0)
	libirc.a (6)	libmbedcrypto.a (8)	libirc.a (1)	std::sync (0)
Total (DV)	17	31	11	2
Code Pointer	libsgx_trts.a (0)	liboecore.a (1)	libenclave.a (0)	std::sys (0)
	libtlibc.a (0)	liboeenclave.a (0)	libsgx_trts.a (0)	std::panicking (0)
	libunwind.a (1)	liboelibc.a (0)	libtlibc.a (0)	std::thread (0)
	libcpprt.a (0)	liboesyscall.a (0)	libunwind.a (1)	std::sys_common (0)
	libirc.a (0)	libmbedcrypto.a (0)	libirc.a (0)	std::sync (0)
Total (CP)	1	1	1	0
Total	18	32	12	2
Function Distribution				
Libraries	libsgx_trts.a (8)	liboecore.a (34)	libenclave.a (2)	std::sys (1)
	libtlibc.a (6)	liboeenclave.a (1)	libsgx_trts.a (63)	std::panicking (2)
	libunwind.a (4)	liboelibc.a (11)	libtlibc.a (73)	std::thread (1)
	libcpprt.a (3)	liboesyscall.a (4)	libunwind.a (12)	std::sys_common (1)
	libirc.a (5)	libmbedcrypto.a (18)	libirc.a (16)	std::sync (1)
				alloc::sync (1)
Total	26	68	166	7

Table 4.2: Data race detection results for the four SGX SDKs

`sgx_thread_mutex_lock` and thus `global_mutex_0` and `global_mutex_1` are generated API defined lock variables, and they are live at line 3-5 for `global_mutex_0`, and line 10 for `global_mutex_1` because they are killed at line 6 and line 11, respectively.

4.3.2.2 Lockset Analysis.

With the liveness analysis of the lock variables, we could have used them to detect the races by intersecting the locksets for the shared variable accesses. If the intersection is empty, it means there is no common lock, resulting in a data race. However, not all shared variables accesses with empty intersection of locksets are data races. For instance, as shown in Figure 4.5, the shared variable access at line 7 holds a lockset of {global_mutex_0} and the access at line 15 holds a lockset of {global_mutex_1}. The intersection of the two locksets is an empty set, but in fact there is no data race between them. Intuitively, to reach line 7, thread 0 must have acquired `global_mutex_0`, which prevents thread 1 from acquiring it in line 12, and thus line 7 and line 15 cannot execute in parallel and no data race can occur. Therefore, besides the locksets, we also need to consider the history of when and

which locks are acquired and released, and a *lock acquisition history* is thus needed to detect the data races. This is essentially how the lockset algorithm [51] works.

As such, we have to formally introduce *lockset*, *lock acquisition history*, and *consistent lock acquisition histories*, and they are defined as follows:

Definitioin 1 (Lockset). *Given thread T_i and an instruction I, we define the lockset(T_i, I) to be the possible set of locks alive at instruction I with thread T_i.*

Definitioin 2 (Lock Acquisition History) *Given thread T_i and an instruction I, for lock l, if $l \in$ lockset(T_i, I), then we define the lock acquisition history LockHistory(T_i, l) to be the set of locks that were acquired (and possibly released) by T_i after the last acquisition of l by T_i.*

Definitioin 3 (Consistent Lock Acquisition Histories). *Given two locks, l_0 and l_1, their acquisition histories are consistent if and only if there do not exist locks l_0 and l_1, such that l_0 is in lock acquisition history of l_1 and l_1 is in lock acquisition history of l_0.*

To collect lockset and lock acquisition history, SGX-RACER also relies on the liveness analysis, as shown in line 35-43. In particular, for lockset, each time when a lock is defined, the liveness analysis generates (`GEN`) the lockset (i.e., the lock is added to the lockset), and kills (`KILL`) the lockset (i.e., the lock is removed from it) if the lock is unlocked. As for lock acquisition history, each time when a lock is defined, the liveness analysis kills (`KILL`) the lock history for this lock (set it to be empty), and generates (`GEN`) the lock history for other locks (add it to the lock history). In this way, the liveness analysis is able to generate lockset and lock acquisition history for each instruction. As in our working example, for instance, line 3 has the lockset of {`global_mutex_0`} since `global_mutex_0` is the only lock acquired (at line 2) and not yet released (at line 6). LockHistory(3, {`global_mutex_0`}) is empty, since after the last acquisition of lock {`global_mutex_0`}, no locks are acquired

(and possibly have been released). The lockset and lock acquisition history for each shared variable access are listed in Figure 4.4.

Lockset-based Data Race Detection. Our lockset-based data race detection algorithm (line 44-52) is based on the lockset algorithm in [52], in which for every shared variable access pair, it checks three conditions: (1) whether one of the accesses is a write (line 45-46), and (2) whether there is no common lock in the two locksets (line 47-48), and (3) whether the lock acquisition history for these two instructions are inconsistent (line 49-51). If all of these conditions are met, SGX-RACER marks the shared variable access pair as a data race. For instance, in our working example, column six of Figure 4.4 lists 12 shared variable access pairs that are potential data races. Five of these pairs contains at least a write, eight pairs have empty lock sets intersections, and none of them have consistent lock acquisition history. Therefore, two data races are identified since they satisfy the three required conditions, and these two data races both originated from one shared variable, i.e., `global_counter`. Throughout this paper, the number of data races are reported along with the number of shared variables that contribute to the data races.

4.4 Evaluation

We have implemented SGX-RACER atop `angr` [101], an open source binary analysis framework. Currently, SGX-RACER supports four well known SGX SDKs: Intel SGX SDK, Microsft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK. SGX-RACER uses `angr` to parse SGX enclave binary code and performs variable analysis and data race detection. The source code of SGX-RACER will be made public available at `github.com`.

SGX Applications											
	# Data Races Shared Vars	# Data Races	Variable Analysis (m)	Race Detection (m)	Total Time(m)	# Shared Var. Access	# Shared Var. Uniq. Var.	# Lock Var. Access	# Lock Var. Uniq. Var.	Ave. Lockset	Ave. Acq. History
Cryptography											
mbedtls-SGX [122]	37	105	205.2	2.4	207.6	138	84	68	3	0.189	0
intel-sgx-ssl [44]	138	646	230.4	3.5	234	477	331	136	4	0.15	0.141
TaLoS [106]	132	471	629.4	6.7	636	417	283	63	7	0.276	0.778
Network											
LibSEAL [62]	2	3	138.6	5.4	144	354	352	0	0	0	0
Database											
SGX_SQLite [98]	95	356	38.4	1.2	39.6	280	177	34	1	0.007	0.000
stealthdb [104]	2	4	157.8	6.4	164.4	6	5	38	1	0.017	0.000
Learning											
SGXDeep [45]	7	24	330.6	6.9	337.8	105	96	29	3	0.599	1.145
Others											
hot-calls [39]	0	0	28.8	0.8	29.4	5	5	44	2	0.026	0.000

Table 4.3: Data race detection results for the SGX applications

In this section, we present our evaluation results. In particular, to evaluate SGX-RACER, we crawled 73 real world SGX projects from `github.com`, and we found 29 of them set `TCSnum` to be 1 in their configuration files. In the rest 44 projects, we select the project that has at least 10 stars in `github.com`, yielding only eight projects in total, and we compile compile these eight projects with Intel SGX SDK (version 2.6) to get their final binaries for our evaluation. We also additionally analyzed three other SDK libraries from Open Enclave SDK (7.0.0), Rust-SGX SDK (1.0.8), and Rust EDP SDK (commit dbe1430) to detect whether the SDK implementations contain any data races for the possible ecalls. All of our experiments were carried out on a Dell x86-64 PC with eight Intel Core i7-7700 processors and 32GB memory. The rest of this section is organized as follows: we first present the detailed effectiveness and efficiency results in §4.4.1 and §4.4.3, and then present a few case studies to demonstrate how the detected data races could be exploited in §4.4.2.

4.4.1 Effectiveness

Recall that a *data race* occurs when there are a pair of memory accesses to the same memory location, such that at least one of them is a write, without any synchronization

primitives that dictate one memory access precedes another memory access, we evaluate the effectiveness of SGX-RACER in SGX SDKs and SGX applications.

Detected Data Races in SGX SDKs. We first apply SGX-RACER to detect the data races in the libraries provided by SGX SDKs. Surprisingly, among them, SGX-RACER have detected in total 39 races, 134 races, 27 races, and 6 races in these four SDKs, with 18, 32, 12, and 2 shared variables identified that contribute to these data races, as reported in Table 4.2. Open Enclave SDK has far more data races than the other three SDKs, and we found a key reason is that a shared variable namely `mul_count` (which is a unit test variable left in binary) is involved in 55 data races, consisting 41% of its total detected data races.

To also illustrate how SGX-RACER analyzes these SDKs, we also provided a number of internal statistics, including the number of shared variable accesses, unique shared variables, lock variables accesses, unique lock variables, lockset size, etc., in Table 4.2. In particular, the number of read-only accesses of the shared variable in each SDK are 317, 591, 362, and 161, respectively. The number of write-only accesses are 119, 214, 134, and 21 and the number of read & write accesses are 6, 7, 16, and 7. The number of unique shared variables identified is 143, 197, 138, and 81, respectively. As for lock variable accesses, SGX-RACER identifies 7, 9, 0, and 20 mutex lock variable accesses, 53, 105, 19, and 0 spin lock variable accesses, and 0, 4, 1, and 5 other lock variable accesses in Intel SGX SDK, Open Enclave SDK, Rust-SGX SDK, and Rust EDP SDK, respectively. The unique lock variables identified are 9, 14, 6, and 1 in four SDKs. We also collected the size of lockset and lock acquisition history for each instruction in the four SDKs, as shown in Table 4.2. The maximum size of lockset is 2, 7, 5, and 1 in four SDKs. The minimum size of lockset is 0, 0, 0, and 0 and the average size of the lockset for each instruction are 0.46, 0.36, 0.93, and 0.30. The maximum size of the lock acquisition history is 8, 13, 5, and 0, respectively.

The minimum size of the lock acquisition history is 0, 0, 0, 0 and the average size of the lock acquisition history is 3.34, 0.10, 1.47, and 0.00, respectively.

Finally, to check which libraries these variables and functions belong to, we also manually identified the racing variable and racing functions (shared variable and functions involved in the data race) in each detected data race in the source code of the four SDKs, and list the distributions in Table 4.2. Interestingly, some data races come from third-party libraries, e.g., `libirc.a`, which is a closed-source library and uses global variables without concerning thread safety. We also find that some SDKs even leave their unit test code in binary which involves data races and global variables and those data races have no lock protection. For instance, Open Enclave SDK binary has global variables such as `mul_count` which introduce data races. The testing code left by developers enlarges trusted computing base (TCB) and attack surface, which should be avoided in a trusted execution environment such as Intel SGX. Another interesting finding is that even Rust-SGX reuses some code from Intel SGX SDK, as it alters the compilation configurations, Rust-SGX is compiled to a different binary, which leads to different race detection results.

Detected Data Races in SGX Applications. Next, we tested SGX-RACER with eight open source projects collected from `github.com`. We group these projects into different categories and present the evaluation results in Table 4.3. As shown in this table, SGX-RACER detects in total $1,609$ data races and 413 contributing variables from these eight SGX projects. The SGX application with the maximum number of data races is `intel-sgx-ssl` (which has 646 data races with 138 contributing variables), and also there are one SGX application that has 0 data race due to the properly synchronized shared variable accesses.

Similarly when presenting the internal statistics for the SGX SDKs, we also show the internal statistics for these SGX applications in Table 4.3. We can notice from this table

that the average number of shared variable accesses is 137.4 with the maximum of 477 (`intel-sgx-ssl`) and the average number of unique variables is 53.5 with the maximum of 352 (`LibSEAL`). As for lock variables, the average number of accesses is 51.5 with the maximum of 136 (`intel-sgx-ssl`) and the average number of unique lock variables identified is 2.6 with the maximum of 7 (`TaLoS`). Among the eight SGX projects, the average lockset size is 0.158 with the maximum of 0.599 (`SGXDeep`) and the average lock acquisition history is 0.258 with the maximum of 1.145 (`SGXDeep`).

Analysis of False Positives. To determine whether there are any false positives in our result, we manually inspected the source code of four SGX SDKs and eight SGX application projects. The manual inspection follows the criteria below:

- Whether the two memory accesses are on the same shared variable.
- Whether at least one of the two accesses is a write access.
- Whether the two accesses could happen at the same time.

As a result, in four evaluated SGX SDKs, We find 7, 29, 0, and 0 false positives, rendering a false positive rate of 17.48%. As for the eight SGX projects, we find 16, 7, 0, 3, 0, 1, 0, 0 false positives, with a false positive rate of 1.68%. The overall false positive rate is 3.47%. The false positives are due to two reasons: (1) Initialization routines. For instance, an initialization routine `_GLOBAL__sub_I_tmem_mgmt.cpp` in SGX project `intel-sgx-ssl` is used to initialize the static variable `addr_info_map`, which cannot be execute in parallel with other accesses to this static variable. (2) Dead code. An example in Open Enclave SDK is `mbedtls_ecp_self_test`, which is used for unit testing in the mbedtls library. Even though `mbedtls_ecp_self_test` is compiled into the enclave code, it cannot

be called from the enclave entry. The dead code will not race with other functions. Since SGX SDKs contain more library functions with initialization routines and dead code, SDKs have higher false positive rates than applications.

4.4.2 Security Case Studies

Certainly, not all data races are exploitable. To determine whether a data race is exploitable is a separate problem, and requires the analysis of the security impact for each race. Nevertheless, in the following, we demonstrate with six case studies to show some of these races can be exploited.

Setting two Rust-SGX enclaves with the same `ENCLAVE_ID`**.** In Apache Teaclave Rust-SGX SDK, SGX-RACER detects a data race on shared variable `ENCLAVE_ID` in function `t_global_init_ecall`, as shown in Figure 4.6a. We can exploit this data race by calling `initialize_enclave` in two threads at the same time, which successfully makes two Rust-SGX enclaves have the same `ENCLAVE_ID`, as shown in Figure 4.7a.

Crashing `oe_shm_malloc` **in Open Enclave SDK.** In Open Enclave SDK, SGX-RACER detects a data race on shared variable `capacity` in function `oe_shm_malloc` and `oe_configure_shm_capacity`, as shown in Figure 4.6b. We can set `capacity` to zero in another thread so that further calls to function `oe_shm_malloc` will fail, as shown in Figure 4.7b. This gives the attackers an ability to Denial of Service at the attackers' discretion.

Corrupting function `srand` **seeds and de-randomizing function** `random` **output.** In Open Enclave SDK, SGX-RACER detects a data race on shared variable `seed` in function `srand` and `rand` in the third party musl libc [78], as shown in Figure 4.6c. We keep setting the shared variable `seed` to a chosen value (e.g., `0x0`) and successfully derandomized the rand function's return value in another thread, as shown in Figure 4.7c. This shows a

potential devastating security threat that data races could be used to disable pseudo-random number generation in enclave code.

Changing a callback function from another thread. The code snippet from Open Enclave SDK in Figure 4.6d has a data race on shared variable `_failure_callback`. We let one thread keep updating the failure callback function to `NULL` pointer. Thus, the failure callback function in the second thread will be overwritten to `NULL` and the thread avoids executing the intended callback function when checking its value at line 8, as illustrated in Figure 4.7d. This case shows that a data race could even happen on shared pointers which might be more devastating than non-pointers.

Replacing the built network in deep learning. SGX application intel-sgx-deep-learning [45] has a data race on shared variable `final_net` in function `ecall_build_network`, as shown in Figure 4.6e. The write of shared variable `final_net` at line 9 is improperly synchronized without any lock and an concurrent thread could replace the built network with its own network, which is later used in deep learning.

Corrupting X509 certificate extension decoding. SGX application ToLoS [106] has a data race on shared variable `in_bc` in function `ecall_X509_get_ext_d2i` as illustrated in Figure 4.6f. Function `ecall_X509_get_ext_d2i` is used for decoding X509 certificate extension with a specific OID, which further calls function `X509_get_ext_d2i` at line 4 and assigns the decoding results to shared vairable `in_bc` at line 6 without proper synchronization. A malicious thread could corrupt this buffer, leading to an incorrect decoding.

4.4.3 Efficiency

We also report the performance of how long SGX-RACER took to analyze these benchmarks, in both Table 4.2 and Table 4.3. More specifically, Table 4.2 lists the average processing time of SGX-RACER in four SGX SDKs. The total processing time is 509.2 minutes, 12616.6 minutes, 453.8 minutes, and 441.4 minutes for four SDKs, in which data race detection phase takes 0.4 minutes, 2.2 minutes, 0.2 minutes, and 0.2 minutes, respectively. The results show that most of the processing time is used in variable analysis phase, which takes time to analyze the data flows. Table 4.3 lists the time overhead in eight SGX applications, and the average total time used in each application is 224.1 minutes and the maximum total time used is 636.0 minutes (`TaLoS`). The average variable analysis time is 219.9 minutes with the maximum of 629.4 minutes (`TaLoS`) and the average data race detection time is 4.2 minutes with the maximum of 6.9 minutes (`SGXDeep`).

```
pub extern "C" fn t_global_init_ecall(
  id: u64, path: * const u8, len: usize)
{
  enclave::set_enclave_id(
  id as sgx_enclave_id_t);
  let s = unsafe {
    let str_slice = slice::from_raw_parts(
    path, len);
    str::from_utf8_unchecked(str_slice)
  };
  enclave::set_enclave_path(s);
}
```

(a) A data race on ENCLAVE_ID in Apache Teaclave Rust-SGX SDK

```
bool oe_configure_shm_capacity(
  size_t cap) {
  ...
  capacity = cap;
  return true;
}

void* oe_shm_malloc(size_t size) {
  ...
  shm.capacity = capacity;
  ...
}
```

(b) A data race on capacity in Open Enclave SDK

```
static uint64_t seed;


void srand(unsigned s) {
  seed = s-1;
}


int rand(void) {
  seed = 6364136223846793005ULL*seed + 1;
  return seed>>33;
}
```

(c) A data race on seed in Open Enclave SDK

```
static oe_allocation_failure_call-
back_t _failure_callback;
void oe_set_allocation_failure_call-
  back(oe_allocation_failure_call-
  back_t function) {
  _failure_callback = function;
}
void* oe_malloc(size_t size) {
  ...
  if (_failure_callback)
    _failure_callback(__FILE__,
    __LINE__, __FUNCTION__, size);
  ...
}
```

(d) A data race on _failure_callback in Open Enclave SDK

```
network *final_net;
void ecall_build_network(char *file_string,
  size_t len_string,
  char *weights, size_t size_weights) {
  ...
  network *net = (networ *)malloc(
    sizeof(network));
  list *sections = sgx_file_string_to_list(
    file_string);
  net = sgx_parse_network_cfg(sections);
  ...
  final_net = net;
  ...
}
```

(e) A data race in SGXDeep

```
BASIC_CONSTRAINTS *in_bc = NULL;


void * ecall_X509_get_ext_d2i(X509
  *x, int nid, int *crit, int *idx)
{
  void* ret = X509_get_ext_d2i(x,
    nid, crit, idx);
  ...
  in_bc = (BASIC_CONSTRAINTS*)ret;
  out_bc->ca = in_bc->ca;
  out_bc->pathlen = in_bc->pathlen;
  ...
}
```

(f) A data race in TaLoS

Figure 4.6: The code snippet in our security case studies

```
void* thread_function(void* data) {
  initialize_enclave();
}
int SGX_CDECL main(int argc,
  char *argv[]) {
  ...
  /* try to create two threads and
     initialize at the same time, will
     trigger data race (based on
     thread interleaving) */
  pthread_create(&threads[0], NULL,
  thread_function, (void *)data);
  pthread_create(&threads[1], NULL,
  thread_function, (void *)data);
  ...
}
```

(a) PoC for the data race on ENCLAVE_ID.

```
void ecall_thread_1()
{
    // malloc
    oe_shm_malloc(1000);
}

void ecall_thread_2()
{
    /* set shm capacity
       to zero */
    oe_configure_shm_capacity(0);
}
```

(b) PoC for the data race on capacity.

```
void ecall_thread_1()
{
  /* keep setting the seed to a
     constant value */
  while(1)
    srand(100000000000000);
}


void ecall_thread_2()
{
  while(1)
    oe_printf("rand␣returns:␣%d\n",
      rand());
}
```

(c) PoC for the data race on seed.

```
void unexpected_callback() {
  oe_printf("unexpected_callback
␣␣triggered\n");
}
void ecall_thread_1() {
  /* set the failure callback to
     another callback function */
  oe_set_allocation_failure_callback(
  unexpected_callback);
}
void ecall_thread_2() {
  // trigger the failure  callback
  oe_malloc(
  10000000000000000000000000000);
}
```

(d) PoC for the data race on _failure_callback.

Figure 4.7: Proof-of-concept (PoC) code for exploiting detected data races.

4.5 Summary

We have presented SGX-RACER, which detects data races in enclave code, by systematically exploring possible concurrent enclave ecalls from both intended and unintended thread interleavings. We have implemented SGX-RACER and evaluated it with eight open source SGX applications and four SGX SDKs, with which SGX-RACER has identified totally 1,582 data races from 405 contributing shared variables in SGX applications, and 32 data races from 14 shared variables, 105 data races from 24 shared variables, 27 data races from 12 shared variables, and 6 data races from 2 shared variables in Intel SGX SDK, Open Enclave SDK, and Rust-SGX SDK, respectively. Such a high alarming number shows that data race in SGX enclave is a serious problem, and a detection tool such as SGX-RACER will be useful to help both developers and the community at large identify and fix them.

Chapter 5: Exploring Value Set Analysis for Capturing Program Semantics in Binary Similarity Analysis

5.1 Overview

5.1.1 Problem Statement

As summarized in Table 5.1, existing efforts on binary similarity search mostly focus on within either the same architecture, or the same platform, and they largely focus on tackling the cross-optimization challenges. Also, most of them focus on structural approaches. In this work, instead, *we aim to tackle the problem of given two binaries, regardless of their architectures, platforms, compilers, and compiler-optimizations, whether there is any similar code between them*. This is the most difficult problem and we must explore architectural-neutralized, platform-independent, compiler-agnostic, and optimization-resilient approaches for cross binary search.

5.1.2 Challenges

Papers	Year	CA	CP	CO	VB	SB	SA	DA
Gao et al. [34]	2008			●	●	●	●	●
Hu et al. [40]	2009					●	●	
Wang et al. [115]	2009			●		●		●
Sæbjørnsen et al. [95]	2009			●		●	●	
Jhi et al. [50]	2011			●	●			●
Chaki et al. [11]	2011					●	●	
Zhang et al. [123]	2012			●	●	●		●
Khoo et al. [56]	2013			●		●	●	
David et al. [19]	2014					●	●	
Luo et al. [71]	2014			●	●	●	●	
Egele et al. [28]	2014			●	●			●
Pewny et al. [85]	2015	●		●	●	●	●	
David et al. [16]	2016			●	●		●	
Chandramohan et al. [12]	2016	●	●	●		●	●	
Su et al. [105]	2016				●	●		●
Feng et al. [31]	2016	●		●		●	●	
Eschweiler et al. [30]	2016	●		●		●	●	
Ding et al. [24]	2016					●	●	
David et al. [17]	2017	●		●		●	●	
Xu et al. [118]	2017	●		●		●	●	
Ming et al. [75]	2017			●	●		●	●
Wang et al. [114]	2017			●	●	●		●
David et al. [18]	2018	●		●		●	●	
Gao et al. [35]	2018			●	●	●	●	●
Liu et al. [66]	2018	●		●		●	●	
Ding et al. [23]	2019			●		●	●	
Zuo et al. [126]	2019	●		●		●	●	
Duan et al. [27]	2020			●		●	●	
vDiff	2021	●	●	●	●		●	

Table 5.1: Comparison of existing binary code search works.

C1: Binary syntax differs substantially on different architectures. Nowadays software programs such as IoT firmware are distributed on various platforms and architectures. Binaries are thus compiled for different architectures and are different in their instruction sets, calling conventions, etc. For instance, Figure 5.1 is a simple example illustrating different assembly code generated from the same source code (Figure 5.1a) for five different architectures. In particular, instructions `ret` and `retq` in x86 and x86-64 return to the address specified on the stack, while instructions `bx`, `ret` and `jr` in ARM, AArch64, and MIPS return to the address specified in a register, which is `lr`, `x30` and `ra` in the example, respectively. When

```
char arr[128];
void encrypt()
{
 int i, num1 = 0xaa, num2 = 0xbb;
 for(i = 0; i < 0x80; i++){
  if(i < 0x40) arr[i] ^= num1;
  else arr[i] ^= num2;
 }
}

int hash()
{
 int i, value = 0, num = 0xcc;
 for(i = 0x60; i > 0x20; i--){
  arr[i] = (arr[i%8]+num) % 0x10;
  value += arr[i];
 }
 return value;
}

void salt()
{
 int i;
 int num[]={0x10,0x20,0x30,0x40};
 for(i = 0x40;i < 0x80;i++){
  arr[i] = num[i % 4];
 }
}
```

(a) Source code

```
8048430 <encrypt>:
8048430:xor %eax,%eax
8048432:jmp 8048449
8048438:xorb $0xaa,0x8049820(%eax)
804843f:add $0x1,%eax
8048442:cmp $0x80,%eax
8048447:je 804845f
8048449:cmp $0x3f,%eax
804844c:jle 8048438
804844e:xorb $0xbb,0x8049820(%eax)
8048455:add $0x1,%eax
8048458:cmp $0x80,%eax
804845d:jne 8048449
804845f:repz ret
```

(b) x86

```
400620 <encrypt>:
400620:xor %eax,%eax
400622:jmp 40063b
400628:xorb $0xaa,0x601060(%rax)
40062f:add $0x1,%rax
400633:cmp $0x80,%rax
400639:je 400654
40063b:cmp $0x3f,%rax
40063f:jbe 400628
400641:xorb $0xbb,0x601060(%rax)
400648:add $0x1,%rax
40064c:cmp $0x80,%rax
400652:jne 40063b
400654:repz retq
```

(c) x86-64

```
84ac <encrypt>:
84ac:movw r3, #0x96ec
84b0:movs r2, #0
84b2:movt r3, #0
84b6:ldrb r1, [r3, #0]
84b8:cmp r2, #0x3f
84ba:add.w r2, r2, #1
84be:add.w r3, r3, #1
84c2:ite le
84c4:eorle.w r1, r1, #0xaa
84c8:eorgt.w r1, r1, #0xbb
84cc:cmp r2, #0x80
84ce:strb.w r1, [r3, #-1]
84d2:bne.n 84b6
84d4:bx lr
```

(d) ARM

```
8d8 <encrypt>:
8d8:adrp x2, 0x10000
8dc:mov x0, #0x0
8e0:mov w5, #0xffffffbb
8e4:mov w4, #0xffffffaa
8e8:ldr x2, [x2, #0xfd8]
8ec:mov x3, x2
8f0:b 910
8f8:ldrb w1, [x0, x3]
8fc:eor w1, w1, w4
900:strb w1, [x0, x3]
904:add x0, x0, #0x1
908:cmp x0, #0x80
90c:b.eq 930
910:cmp x0, #0x3f
914:b.ls 8f8
918:ldrb w1, [x0, x2]
91c:eor w1, w1, w5
920:strb w1, [x0, x2]
924:add x0, x0, #0x1
928:cmp x0, #0x80
92c:b.ne 910
930:ret
```

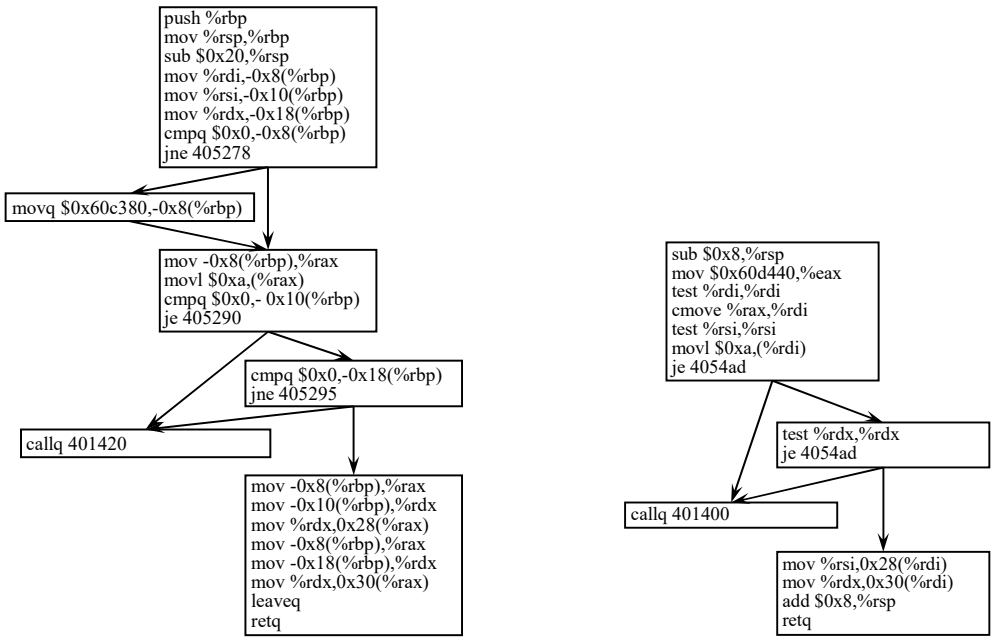
(e) AArch64

```
7b0 <encrypt>:
7b0:lui gp,0x2
7b4:addiu gp,gp,-32144
7b8:addu gp,gp,t9
7bc:lw a0,-32732(gp)
7c0:move a1,zero
7c4:li t0,-69
7c8:sltiu v1,a1,64
7cc:li a3,127
7d0:li a2,-86
7d4:beqz v1,7f8
7d8:lb v0,0(a0)
7dc:xor v0,v0,a2
7e0:sb v0,0(a0)
7e4:addiu a1,a1,1
7e8:lb v0,1(a0)
7ec:sltiu v1,a1,64
7f0:bnez v1,7dc
7f4:addiu a0,a0,1
7f8:xor v0,v0,t0
7fc:bne a1,a3,7e4
800:sb v0,0(a0)
804:jr ra
```

(f) MIPS

Figure 5.1: A simple example illustrating assembly code generated from the same source code across five architectures.

function `encrypt` calculates "`arr[i] ^= num1`", the `xor` instruction accepts two operands in x86 and x86-64, whereas the `eor` and `xor` instructions accept two or three operands in ARM, AArch64, and MIPS. These differences add architectural noises to binary code search. Being a value-based approach, vDIFF needs to identify and neutralize these noises.



(a) Compiled with -O0 (b) Compiled with -O3

Figure 5.2: CFGs of `set_custom_quoting` generated from different compiler optimization levels.

C2: Various compilers and compiler optimizations will greatly change the final binary code. Today, there are a variety of compilers such as GCC and LLVM. Meanwhile, modern compilers often provides many optimizations, *e.g.,* GCC version 5.4.0 provides 215 optimization flags and groups them into optimization levels with -O0, -O1, -O2, -O3, -Ofast, -Og, and -Os for the user's convenience. Although general developers and also prior efforts on binary code search only consider -O3 as the most sophisticated optimization configuration, it only covered 138 optimization flags (for GCC version 5.4.0), which is only 64% of the total optimization flags. It might be possible for a developer to compile a program code using non-default optimization levels (by using optimizations not in -O3). The combination of optimization flags could be 2^{215} theoretically, which makes it extremely hard (or infeasible) for a learning based approach to construct all cases. Note that compiler

optimizations could impose great changes to final binary code. For instance, Figure 5.2a and Figure 5.2b are control flow graphs of function `set_custom_quoting` of GNU `coreutils` compiled by GCC 5.4.0 under -O0 and -O3 optimizations, respectively. The structure of CFG changes significantly and the number of edges and nodes also drops sharply which would greatly hinder structural-analyses such as graph-based similarity analysis.

C3: VSA-signatures need to be properly refined. While using VSA for similarity detection is intuitive (as it can generate a tuple of value sets for a function at the function exit point as a signature for similarity comparison), unfortunately some value sets are architecture noises and should not be involved in similarity comparison. For instance, some value sets are only about stack offsets which involve architectural information, *e.g.,* the value set of $(\perp, [0x2c, 0x2c], \perp)$ indicates that it has an offset of `0x2c` to the current stack, but the offset might be quite different (*e.g.,* $(\perp, [0x400, 0x400], \perp)$) or even missing in a different architecture (*e.g.,* in architectures that prefer less register spilling). vDIFF needs to identify those architecture noises (*e.g.,* value sets representing stack offsets) and tell them apart from architecture-independent value sets that capture the real semantics of a function.

C4: VSA-signatures need to be efficiently compared. Each per-function tuple of value sets generated by VSA has hundreds (or even thousands) of value sets. vDIFF has to address this challenge via an effective way of comparing the value sets (which even might have quite different sizes). Moreover, value sets are not a trivial quantity. Each value set has a different upper bound, lower bound, and stride. Comparing two value sets means an overall comparison over all those dimensions. A value set based binary code search approach needs to effectively compute the similarity between the two value sets which may be quite different and need to collectively compare tuples of value sets based on the similarity of

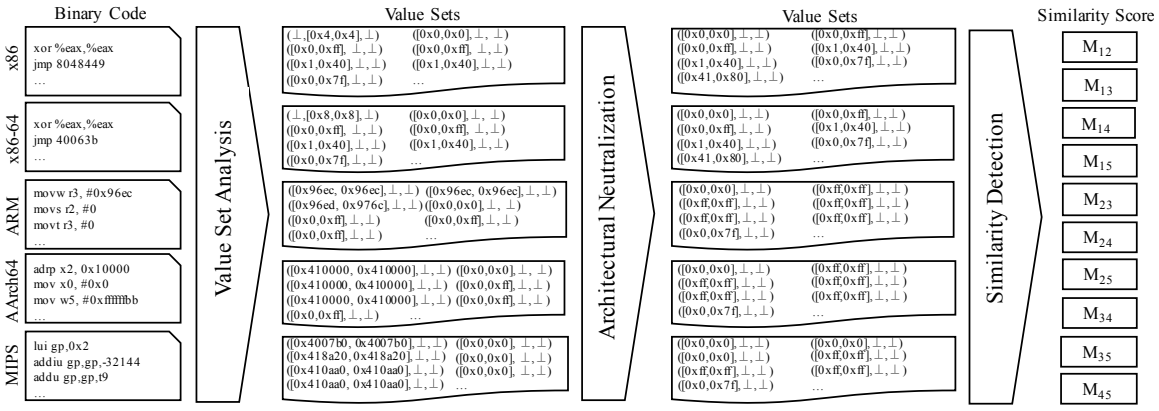


Figure 5.3: Overview of the workflow of VDIFF.

each individual value set to faithfully represent the similarity between the two pieces of binary code from different architectures.

5.1.3 Key Insights

Each of the above challenges can be addressed correspondingly in our VSA-based cross-architecture binary code search approach. In particular, C1 can be addressed via the power of the value set analysis abstractions. When using VSA, the instruction opcode is abstracted as a transfer function which changes the value sets at the exit point of the instruction and the instruction operands are abstracted using a-locs and value sets. As such, the quite different instructions in different syntax across architectures are all turned into uniformly well-defined a-locs and value sets.

For C2, compiler optimizations do add noises, *e.g.,* an instruction calculating "`a = a + 2`" could be split by two `a++` instructions, adding an intermediate value set of constant value 1. However, value set analysis could still reason the final effects as "`a = a + 2`", since compiler optimizations do not alter the semantics and the final refined value sets remain the same. In this way, C2 is mitigated by capturing the final refined value sets in our approach.

Being a value-based approach, C3 is addressed by designing an architectural neutralization step which identifies and eliminates architecture related value sets, such as stack addresses, stack adjustment offsets, and code addresses. Thus, the architectural differences are neutralized, and the remaining refined value sets are considered in comparing value sets tuples for binary similarity analysis.

To solve C4, we first classify different value sets based on their boundedness (*i.e.,* bounded, left-bounded, right-bounded, or unbounded) and we derive individual value set similarity within each category. Then, we count the similar value sets and dissimilar value sets in each category. Finally, by further considering tuple features such as tuple ordering and tuple sizes, a similarity score is generated to faithfully capture the similarity of two functions in different binary code across different architectures.

5.1.4 Overview

An overview of our vDIFF is illustrated in Figure 5.3. There are three phases of analysis inside vDIFF: (*i*) value set analysis (§5.2.1), (*ii*) architectural neutralization (§5.2.2), and (*iii*) similarity detection (§5.2.2). At a high level, the first phase is to perform VSA for each function of the binary code under analysis. Since vDIFF works at function level, it has to first carry out the following pre-processing steps: *function identification* to find function boundaries in binary code, *backward slicing*, and *function signature identification* to establish missing edges in control flow graph, and *function exit identification* to locate function returns or inter-procedural jumps. Then VSA is performed on all identified functions based on the CFG recovered in pre-processing steps, and the value sets generated via VSA on the function are collected for further binary similarity analysis. The collected value sets contain

however architectural information, and those architecture noises are further neutralized and the remaining value sets are used as a signature for the comparison.

Working Example.. In the following, we use the simple program in Figure 5.1 again with a particular focus on x86-64 binary code (Figure 5.1c) and AArch64 binary code (Figure 5.1e) to illustrate how VDIFF performs binary code search across different architectures step by step. The example has three functions `encrypt`, `hash` and `salt`, which work on a global array `arr` containing 128 bytes. Function `encrypt` encrypts `arr` by xor-ing each byte in the array with a number, while function `hash` calculates hashes for `arr` by transforming 64 bytes of the array and function `salt` adds salt to the array by changing the last 64 bytes of the array, as illustrated in Figure 5.1a. We only list the assembly code of function `encrypt` for illustration purpose in Figure 5.1b - Figure 5.1f.

The value set analysis results of function `encrypt` for x86-64 binary and AArch64 binary are listed in Table 5.2 and Table 5.3, respectively. The value sets are denoted as 3-tuples (O^g,O^s,O^h), each element of which is the particular offset of global, stack and heap region, respectively. At the beginning of each binary, an index into the array is assigned 0 as the initial value, as shown in instruction `400620` and `8dc`, respectively. After an unconditional jump, a comparison is made to determine whether the index is greater than $0x3f$ or not (instruction at `0x40063b` and `910`). At this program point, the registers holding the index value have a value set of ([0x0,0x7f], $\perp$, $\perp$). Based on the comparison result, the array element is either xor-ed with $0xaa$ or $0xbb$. In Figure 5.1c, $0xaa$ and $0xbb$ are two immediate values in instruction operands, while in Figure 5.1e, $0xaa$ and $0xbb$ are stored in `w4` and `w5` register and sign-extended. The `w4` and `w5` are holding the value sets of ([0xffffffaa,0xffffffaa], $\perp$, $\perp$) and ([0xffffffbb,0xffffffbb], $\perp$, $\perp$). Then the array element is xor-ed with those numbers and stored back as a byte and has a value set of ([0x0,0xff],

Instruction	Abstract Locations	Value Sets	Arch. Indep.
0x400620	eax	([0x0,0x0], ⊥, ⊥)	✓
0x400628	(0x601060, ⊥, ⊥) . . . (0x60109f, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
0x40062f	rax	([0x1,0x40], ⊥, ⊥)	✓
0x400633	rax	([0x1,0x40], ⊥, ⊥)	✓
0x40063b	rax	([0x0,0x7f], ⊥, ⊥)	✓
0x400641	(0x6010a0, ⊥, ⊥) . . . (0x6010df, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
0x400648	rax	([0x41,0x80], ⊥, ⊥)	✓
0x40064c	rax	([0x41,0x80], ⊥, ⊥)	✓
0x400654	rsp	(⊥, [0x8,0x8], ⊥)	✗

Table 5.2: VSA of the x86-64 binary of the working example.

⊥, ⊥) since the bytes stored in the global array `arr` is unknown and the result could be any value within a byte. The index is later increased and has a value set of ([0x1,0x80], ⊥, ⊥) and the loop terminates when it reaches $0x80$.

In each iteration of our VSA, the value set of every a-loc accessed in each instruction is merged with newly generated value set in this iteration to represent the possible values an a-loc has. For instance, in the working example of the x86-64 binary, instruction `40062f` has an a-loc: register `rax`. In the first iteration, a-loc `rax` has a value set of ([0x1,0x1], ⊥, ⊥) and in the second iteration it is ([0x2,0x2], ⊥, ⊥). The two are merged into ([0x1,0x2], ⊥, ⊥) after the second iteration of our analysis and further upcoming value sets are merged the same way until a fixed point is reached. Then all value sets generated for every instruction and its corresponding a-locs are collected as an unordered tuple for the next phase analysis.

The second phase is architectural neutralization, which neutralizes value sets generated by VSA via finding architectural noises and then removing them. As in our working example, in Table 5.2 and Table 5.3, the 4th-column presents whether the corresponding final value set is architectural dependent or not. Clearly, value sets such as (⊥, -0x8, ⊥) in

Instruction	Abstract Locations	Value Sets	Arch. Indep.
0x8d8	x2	([0x410000,0x410000], ⊥, ⊥)	✗
0x8dc	x0	([0x0,0x0], ⊥, ⊥)	✓
0x8e0	w5	([0xffffffbb,0xffffffbb], ⊥, ⊥)	✓
0x8e4	w4	([0xffffffaa,0xffffffaa], ⊥, ⊥)	✓
0x8e8	([0x410fd8,0x410fd8], ⊥, ⊥)	([0x411040,0x411040], ⊥, ⊥)	✗
	x2	([0x411040,0x411040], ⊥, ⊥)	✗
0x8ec	x3	([0x411040,0x411040], ⊥, ⊥)	✗
	x2	([0x411040,0x411040], ⊥, ⊥)	✗
0x8f8	(0x411040, ⊥, ⊥) . . . (0x41107f, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
	w1	([0x0,0xff], ⊥, ⊥)	✓
0x8fc	w1	([0xffffff00,0xffffffff], ⊥, ⊥)	✓
	w4	([0xffffffaa,0xffffffaa], ⊥, ⊥)	✓
0x900	(0x411040, ⊥, ⊥) . . . (0x41107f, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
	w1	([0xffffff00,0xffffffff], ⊥, ⊥)	✓
0x904	x0	([0x1,0x40], ⊥, ⊥)	✓
0x908	x0	([0x1,0x40], ⊥, ⊥)	✓
0x910	x0	([0x0,0x7f], ⊥, ⊥)	✓
0x918	(0x411080, ⊥, ⊥) . . . (0x4110bf, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
	w1	([0x0,0xff], ⊥, ⊥)	✓
0x91c	w5	([0xffffffbb,0xffffffbb], ⊥, ⊥)	✓
	w1	([0xffffff00,0xffffffff], ⊥, ⊥)	✓
0x920	(0x411080, ⊥, ⊥) . . . (0x4110bf, ⊥, ⊥)	([0x0,0xff], ⊥, ⊥)	✓
	w1	([0xffffff00,0xffffffff], ⊥, ⊥)	✓
0x924	x0	([0x41,0x80], ⊥, ⊥)	✓
0x928	x0	([0x41,0x80], ⊥, ⊥)	✓

Table 5.3: VSA of AArch64 binary of the working example.

`rsp` are stack addresses and are architectural noises. And values representing code and data addresses are also architectural noises. Such examples include value set (⊥, [0x4,0x4], ⊥) in Table 5.2 as it represents a stack address, and value sets ([0x410000,0x410000], ⊥, ⊥) and ([0x411040,0x411040], ⊥, ⊥) in Table 5.3 as they represent global addresses. Thus, in our neutralization phase, we identify all those architectural noises and remove them from our final value sets. After this refinement, value sets are grouped into an unordered tuple as our vectorized representation for this function. For x86-64 and AArch64 binary code in our working example, the two unordered tuples (denoted as T_0 and T_1) are:

T_0: (([0x0,0x0], $\perp$, $\perp$), ([0x0,0xff], $\perp$, $\perp$) ... ([0x0,0xff], $\perp$, $\perp$),
([0x1,0x40], $\perp$, $\perp$), ([0x1,0x40], $\perp$, $\perp$), ([0x0,0x7f], $\perp$, $\perp$),
([0x41,0x80], $\perp$, $\perp$), ([0x41,0x80], $\perp$, $\perp$))

T_1: (([0x0,0x0], $\perp$, $\perp$), ([0x0,0xff], $\perp$, $\perp$) ... ([0x0,0xff], $\perp$, $\perp$),
([0xffffffbb,0xffffffbb], $\perp$, $\perp$), ([0xffffffbb,0xffffffbb], $\perp$, $\perp$),
([0xffffffaa,0xffffffaa], $\perp$, $\perp$), ([0xffffffaa,0xffffffaa], $\perp$, $\perp$),
([0xffffff00,0xffffffff], $\perp$, $\perp$), ([0xffffff00,0xffffffff], $\perp$, $\perp$),
([0xffffff00,0xffffffff], $\perp$, $\perp$), ([0xffffff00,0xffffffff], $\perp$, $\perp$),
([0x1,0x40], $\perp$, $\perp$), ([0x1,0x40], $\perp$, $\perp$), ([0x0,0x7f], $\perp$, $\perp$),
([0x41,0x80], $\perp$, $\perp$), ([0x41,0x80], $\perp$, $\perp$))

The value sets come from: values read/written from/to memory cells in heap, stack, global region, or registers. Note that VSA is flow-sensitive and per-instruction. We are particularly interested in the value sets for each executed instruction, since *they represent the whole computed semantic values for the function*. Back to our working example, we will collect the value sets encountered for each instruction from the entry point to the exit point of the function. The lower bound of the number of the VSA a function could have depends on the number of identified memory addresses and registers encountered during the analysis of the function.

The third phase is to calculate the binary code (at function level) similarity based on pairwise similarity metrics. The similarity metrics calculate a score based on how many value sets are the same for two compared functions. The rationale is *the more value sets in common for the two functions, the more similar are for them*. As for the working example, for instance, for the above two tuples (*i.e.,* T_0 and T_1), T_0 has 134 value sets in common with T_1, *i.e.,* as shown in Table 5.2, 1 value set of ([0x0,0x0], $\perp$, $\perp$) which is also in T_1, 128 value sets of ([0x0,0xff], $\perp$, $\perp$), 2 value sets of ([0x1,0x40], $\perp$, $\perp$), 1 value set of ([0x0,0x7f], $\perp$, $\perp$) and 2 value sets of ([0x41,0x80], $\perp$, $\perp$) and therefore $1+128+2+1+2=134$. T_1 has 272 ($1+2+2+258+4+2+1+2=272$) value sets in common with T_0. These counts for

the two tuples are fed into a formula (which is described in §5.2.3) to generate a similarity score. Assume a given binary has N function, and there are M architectures. The similarity metrics for each function among these architectures are calculated pairwise, resulting $C^2_{N\cdot M}$ similarity scores. For each binary, the similarity scores are sorted and functions with the top score are considered similar functions.

5.2 Design and Implementation

5.2.1 Value Set Analysis

Pre-processing.. To generate value sets for our similarity analysis, we first perform multiple pre-processing steps, and this includes function identification, CFG reconstruction, and function exits identification. The function identification is based on `angr` [101] which generates a function list with instructions identified for each function. Then we reconstruct control flow graph (CFG) for each identified function based on the given binary and resolving the possible control flow targets using the Ramblr [112] approach. After reconstructing CFG, inter-procedural control flow transfers are identified, such as inter-procedural jumps, inter-procedural calls and returns, which is later used in value set analysis.

Value Set Analysis.. Based on the CFG generated from pre-processing, vDIFF performs an intra-procedural VSA since this could form a more efficient binary similarity analysis than inter-procedural VSA and meanwhile those computed value sets within a procedural can already form strong signatures (according to our experimental results).

The VSA in vDIFF is described in algorithm 6. Specifically, the initial value sets for the entry instruction is initialized with an initial stack pointer, an initial empty heap, and an initial memory cell value of $(\top,\top,\top)$ (line 53). The main part of the algorithm (line 53-59) is of work list style with multiple iterations on each instruction until a fixed point is reached.

Algorithm 6 The Value Set Analysis in VDIFF

Function *VSA(CFG)*:
    input : control flow graph *CFG*, value set *ValueSet*, function *func*, *context*
    output : ValueSet[i] for each instruction i
53 ValueSet = Init()
    worklist = {entryInst}
    while *worklist* $\neq \emptyset$ **do**
54   i $\leftarrow$ pop(worklist)
55   **if** *condInst(i)* **then**
56          $\text{ValueSet}^i_{exit_n} = \text{ValueSet}^i_{entry} \sqcap^{VS} \text{ValueSet}_{c_n}$
57   **else**
58          $\text{newValueSet}^i_{exit} = \boldsymbol{EXE}(\bigsqcup_{entry_n \in entry} \text{ValueSet}^i_{entry_n})$
            if $newValueSet^i_{exit} \neq ValueSet^i_{exit}$ **then**
59              $\text{ValueSet}^i_{exit} \leftarrow \text{ValueSet}^i_{exit} \sqcup \text{newValueSet}^i_{exit}$
                push(worklist, succs(i))
60 **for** *each* $i \in Inst$ **do**
61      $\text{ValueSet}^i_{changed} \leftarrow \text{changed}(\text{ValueSet}^i)$
        record($\text{ValueSet}^i_{changed}$)

Each iteration handles one instruction based on the type of the instruction. Particularly, if the instruction `i` is a conditional jump, the value set at the exit point of the instruction is further confined with the corresponding path constraint (line 55-56). Otherwise, the value set at the exit point of the instruction is generated based on the semantics of the instruction, which is used as a transfer function in VSA (line 57-58).

As VSA is flow-sensitive and per-instruction, it is nontrivial to capture the impact of each instruction as a transfer function. We thus leverage the semantics of each instruction to implement our VSA transfer function. As shown in line 58 of algorithm 6, all incoming value sets are merged on a per register and memory cell basis and are fed into the transfer function to generate output value sets for each instruction.

After analyzing an instruction, the value sets of each register and memory cell could be changed based on the semantics of the instruction. Next, we check if the value sets are

changed by comparing with earlier value sets; if so, the newly-calculated value sets are merged with the earlier value sets and the successors of this instruction is pushed into the work list for further iterations (line 58-59). Note that since our VSA is an intra-procedural analysis, calls to other functions return at once with fresh unconstrained value sets, and it stops further propagating of value sets when encountering returns or inter-procedural jumps.

After the work list algorithm finishes, VDIFF collects the final value sets generated by VSA for each instruction in the function (line 60-61). Since our algorithm aims to collect the value sets influenced by the current function, we only collect the value sets that are changed during VSA, *i.e.,* if the value set of an abstract location remains unchanged, we consider it is not influenced by the current function (line 61).

The collected value sets are stored in an unordered tuple as our vectorized representation for this function. Our value sets collection has the following two considerations: (1) we do not collect the specific location and specific changes, since across different architectures and different compiler optimizations the location for storing a particular data object can be different, even between registers and memory cells; (2) we do not put value sets in a set which otherwise will remove the duplicated value sets, since different abstract locations can surely have the same value sets and we would like to count those variables even though they have the same value sets.

5.2.2 Architectural Neutralization

After collecting the value sets of our interest at the function exit, VDIFF neutralizes these value sets by removing architectural related ones. The value sets in collected unordered tuples are either addresses or quantities, and only address related ones need to be neutralized. In particular, these address related value sets including the following three categories:

- **Stack address.** Stack addresses are introduced when a program adjusts the stack pointer or access variables stored on the stack. While some architecture favors stack, other architecture may prefer registers. Thus, when analyzing binaries compiled from the same source code for different architectures, a variable could be stored in certain stack address in some architecture, whereas the same variable may be allocated in registers in another architecture. For example in the working example x86-64 binary, ($\perp$, [-0xc,-0xc], $\perp$) and ($\perp$, [-0x10,-0x10], $\perp$) represent stack addresses and might be changed in different architectures and thus we remove them in our final value set tuples for this function.

- **Heap address.** Heap addresses are subject to changes in different architectures, and are different even across different runs of the same binary. For instance, heap allocations are done each time at runtime and thus the start address of a heap allocation is unknown (though we can assign a static symbol address). Also, due to the different word size of each architecture, the data object within a heap allocation may have different offset. Therefore, heap addresses are architecture specific which involves runtime information and the addresses do not contribute to the semantics of the function in binary code. For instance, a value set of ($\perp$, $\perp$, [0x4,0x4]) is neutralized in this phase as it is subject to changes in different architectures.

- **Global address.** Global variable address could be `.data` segment address which contains initialized variables, `.bss` segment address which holds uninitialized variables, or `.text` segment address which has executable instructions. The layout of the final binary totally depends on architecture specific details, compiler optimizations and compiler backends and are thus architecture specific. For example, assume `.data`

segment starts with address `0x400100` and ends with address `0x400300`. A value set of ([`0x400200`,`0x400200`], $\perp$, $\perp$) is removed as the binary layout depends on the architecture it is compiled. Addresses that is not directly within a global address, e.g., `0x400000`, could also be used by the program to calculate variable addresses. These addresses are architectural dependent and should be removed. However, distinguishing integer value and global address (*i.e., symbolization*) is non-trivial and we take Ramblr [112] approach, which recognizes an integer to be a global address if the integer falls within a slightly enlarged global memory region (e.g., 4KB). And thus `0x400000` is within the enlarged `.data` segment starting at `0x3ff100` and ending at `0x401300` and gets removed in architectural neutralization phase.

These address value sets depend on either architecture specifications or compiler implementations, and therefore not suitable for capturing the semantics of the binary. vDIFF neutralizes the architectural features by eliminating the address value sets in the above three categories. After the neutralization, the unordered tuple has fewer elements yet still represents the semantics of the binary code.

Handling word size differences and sign bit.. Since vDIFF deals with both 32-bit and 64-bit architectures, the value representation can appear different for the same value (e.g., `-1` is represented as `0xffffffff` in 32 bit machine, and `0xffffffffffffffff` in 64 bit machine). Therefore, we have to properly normalize them for the comparison.

In particular, if the binaries have the same word size (e.g., both 32 or 64), we still compare them as usual. Only when one of them is 32-bit, then we will extend all of the 32-bit value to 64-bit by using the standard sign-extension algorithm, which increases the number of bits of a binary number while preserving the number's sign (positive/negative) and value. Unfortunately, we face another challenge here depending on the number to be

extended is signed or unsigned. For signed numbers such as -1 and -2, using the standard sign-extension algorithm incurs no problem. However, for unsigned number, when extending them, e.g., UINT32_MAX in 32-bit binaries, vDIFF will sign extend it to a different number UINT64_MAX, which leads to a different value. In contrast, if it is unsigned, we should have used the zero-extension instead of sign-extension.

Therefore, ideally, if we can know the signedness of the number, we can correspondingly perform either sign-extension or zero-extension to convert the 32-bit to 64-bit. However, at the binary code level, it is challenging to perform such inference. For instance, in x86, only few instructions such as `jg` or `ja` can possibly reveal the signedness of values. While we wish vDIFF is able to perform such inference, currently we ignore such analysis and instead we perform both sign-extension and zero-extension for the same 32-bit number. If we find a match in the corresponding 64-bit machine for either value, we regard it as a match.

5.2.3 Similarity Detection

Based on the unordered tuples generated from the previous phases, the similarity detection phase converts pairs of unordered tuples into a similarity score to measure how similar two functions are in binary code.

vDIFF uses Jaccard index to compare similarity. Jaccard index is a statistic used in calculating the similarity of two sample sets. Based on the unordered tuples T_0 and T_1 in our working example, the corresponding sets s_0 and s_1 are:

s_0: ([0x0,0x0], $\perp$, $\perp$), ([0x0,0xff], $\perp$, $\perp$),
([0x1,0x40], $\perp$, $\perp$), ([0x0,0x7f], $\perp$, $\perp$),
([0x41,0x80], $\perp$, $\perp$))

When comparing two sets s_0 and s_1, Jaccard index can be denoted as:

s_1: (([0x0,0x0], ⊥, ⊥), ([0x0,0xff], ⊥, ⊥),
([0xffffffbb,0xffffffbb], ⊥, ⊥),
([0xffffffaa,0xffffffaa], ⊥, ⊥),
([0xffffff00,0xffffffff], ⊥, ⊥),
([0x1,0x40], ⊥, ⊥), ([0x0,0x7f], ⊥, ⊥),
([0x41,0x80], ⊥, ⊥))

$$J(s_0, s_1) = \frac{|s_0 \cap s_1|}{|s_0 \cup s_1|}$$

Regarding the working example in Figure 5.1, we thus have:

$$|s_0 \cap s_1| = 5 \quad |s_0 \cup s_1| = 8 \quad J(s_0, s_1) = \frac{5}{8} = 0.625$$

To gain an intuition about vDIFF similarity analysis, we draw a heat map illustrating the similarity results with different architecture binary code of the working example, as shown in Figure 5.4. Listed horizontally from left to right and vertically from top to down in the heat map are the 3 functions, labeled as `encrypt`, `hash` and `salt` with different architectures and compiler optimization levels appended. The darkness of the cubes represents the similarity of the corresponding function pairs. The darker the cube is, the more similar the corresponding functions are. From Figure 5.4, we can observe that vDIFF has clearly identified the similar functions among different architectures.

5.2.4 Implementation

We have implemented vDIFF, which disassembles binary code using binary disassembler `capstone` [92], resolves memory regions and abstract locations [1, 20–22, 89, 99], builds transfer functions for each instruction to capture its semantics, performs value set analysis to generate value set tuples for each function, neutralizes value set tuples, and calculates the similarity scores. Note that while there are a few open source VSA implementations, *e.g.,* VSA in angr [101, 112] and the recent DeepVSA [37], we do not reuse them directly

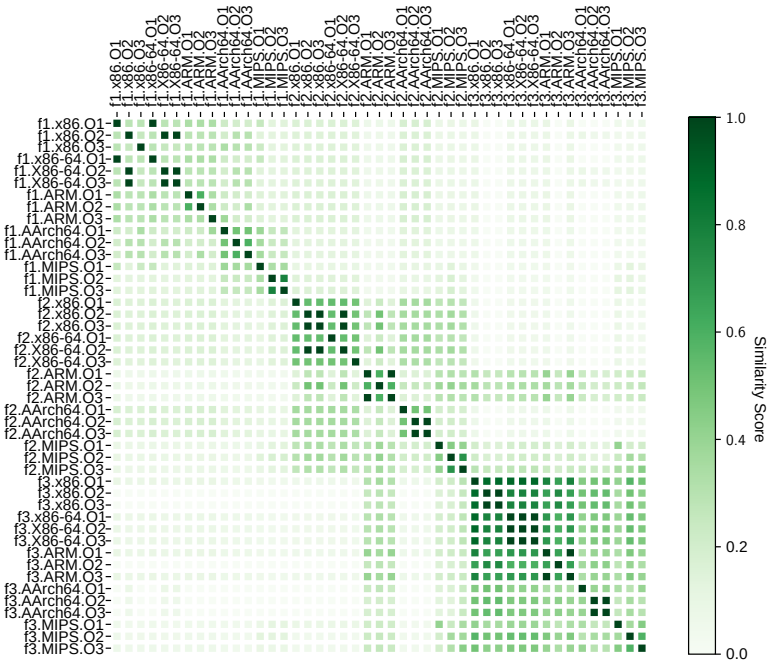


Figure 5.4: Similarity scores generated by VDIFF for the working example across different architectures and optimization levels.

when building VDIFF because they do not satisfy our needs. For instance, angr's VSA does not reason about the abstract locations on the heap. Thus, we build our VDIFF from scratch, which strictly conforms to the concepts of value set analysis and is a pure static analysis framework. VDIFF's implementation comprises 6,000 lines of Python code, and supports binary code search over five architectures: x86, x86-64, ARM, AArch64, and MIPS. In support of open science, the source code of VDIFF will be made public available at `github.com`.

5.3 Evaluation

In this section, we present our evaluation results. We first describe the benchmarks used in our evaluation (§5.3.1), followed by a systematic evaluation with respect to the effectiveness (§5.3.2), in which we evaluate VDIFF when facing cross-architecture binaries.

5.3.1 Experimental Setup

Benchmark I. To evaluate the effectiveness of VDIFF, we use `coreutils` (v8.29) program suite, which we evaluated 107 binaries for each architecture.

All of these programs are compiled with GCC 5.4.0, the default compiler in Ubuntu 16.04, for all five architectures and with three compiler optimization levels O1–O3. This version of GCC has also been used by other related work such as DeepBinDiff [27].

Machine Configuration. Our experiments were conducted on an 8-core 3.60 GHz Intel i7 machine with 32 GB memory, and Ubuntu 16.04 LTS.

5.3.2 Effectiveness

To examine the effectiveness of VDIFF, we run it on a number of architecture, optimization level, and platform pairs. Since we have source code for all the programs in benchmark I and benchmark II, we collect the ground truth using function symbol information. Binary functions f and g are defined as *similar*, written $f \sim g$, when functions f and g were compiled from the *same source code function*.

VDIFF identifies a target function f in architecture α to be the similar function of g in architecture β if it has the largest similarity score with function g among all functions in α. True positives (*TP*) and true negatives (*TN*) occur when VDIFF identifies f as similar to g when $f \sim g$, and identifies f as dissimilar to g when $f \not\sim g$, respectively. Inversely, false

positives (FP) and false negatives (FN) occur when VDIFF misidentifies f and g as similar even though $f \not\sim g$, and misidentifies f and g as dissimilar even though $f \sim g$, respectively. We define precision (P), recall (R), and F1-score (F_1) in the standard ways:

$$P = \frac{TP}{TP+FP} \qquad R = \frac{TP}{TP+FN} \qquad F_1 = 2\frac{P \cdot R}{P+R}$$

Since we only select the function with the largest similarity score for each match, and we have the ground truth, we can calculate VDIFF's overall precision rate and recall rate for all covered functions, each of which has more than a specified value set tuple size (*e.g.,* 10, 100, etc.).

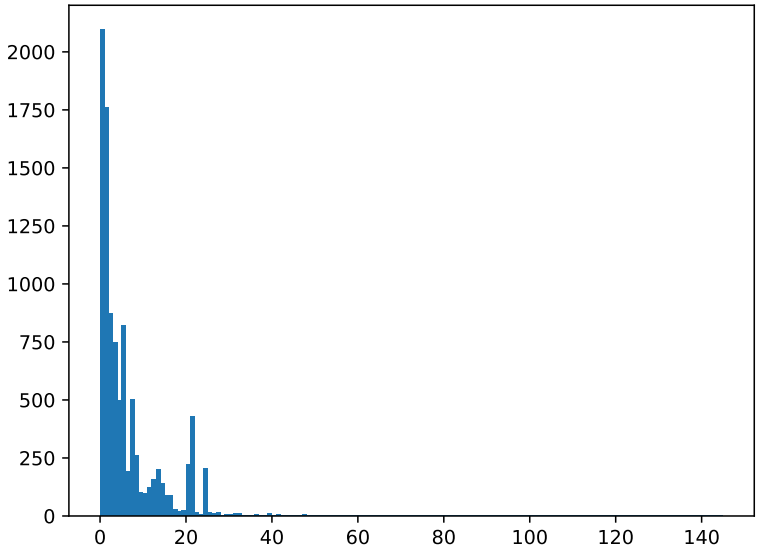


Figure 5.5: Different value set sizes distribution of coreutils x86 binaries.

5.3.2.1 Distribution of value set tuple sizes

Our evaluation focuses on higher-complexity functions, where complexity is measured by value set tuple size, since these are often the functions that pose the greatest challenge for similarity detection. For example, functions with many value sets typically have complex

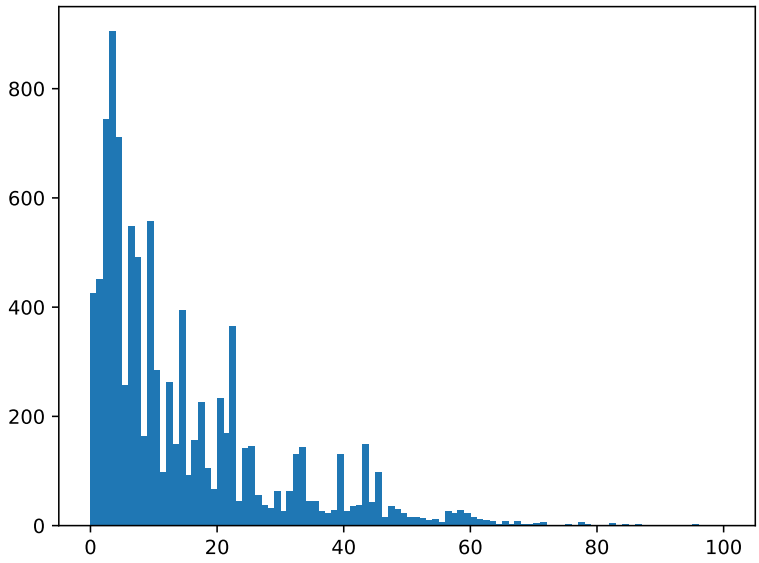


Figure 5.6: Different value set sizes distribution of coreutils x86-64 binaries.

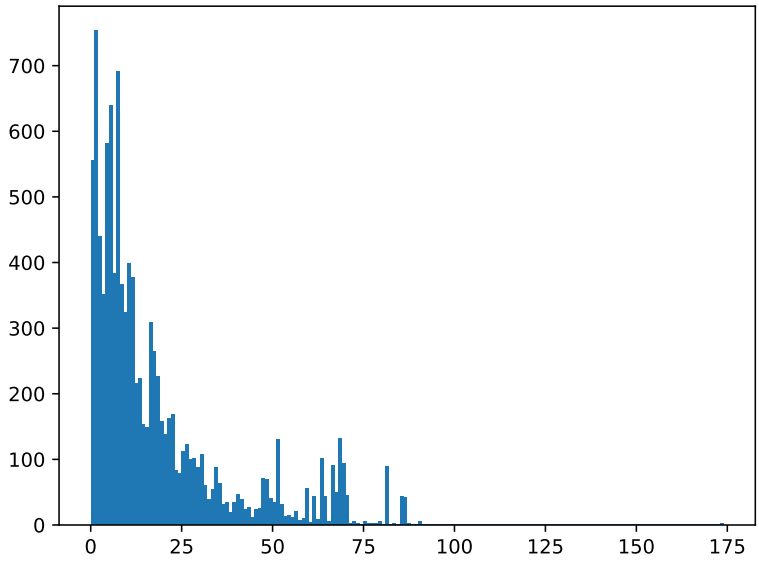


Figure 5.7: Different value set sizes distribution of coreutils ARM binaries.

code structures containing many value assignments, which can raise accuracy problems for structure-based approaches and coverage problems for dynamic approaches. Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, and Figure 5.9 show the ubiquity of such functions by plotting the distribution of functions with different VSA sizes for architecutures x86, x86-64,

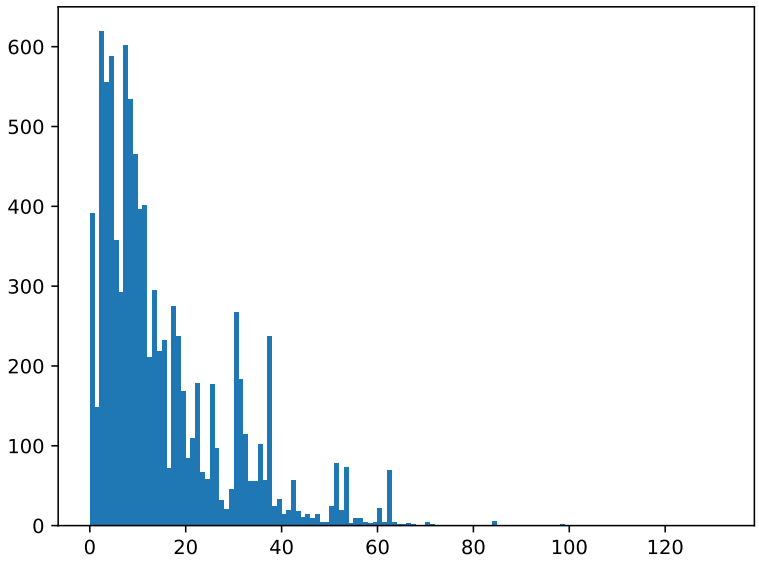


Figure 5.8: Different value set sizes distribution of coreutils AArch64 binaries.

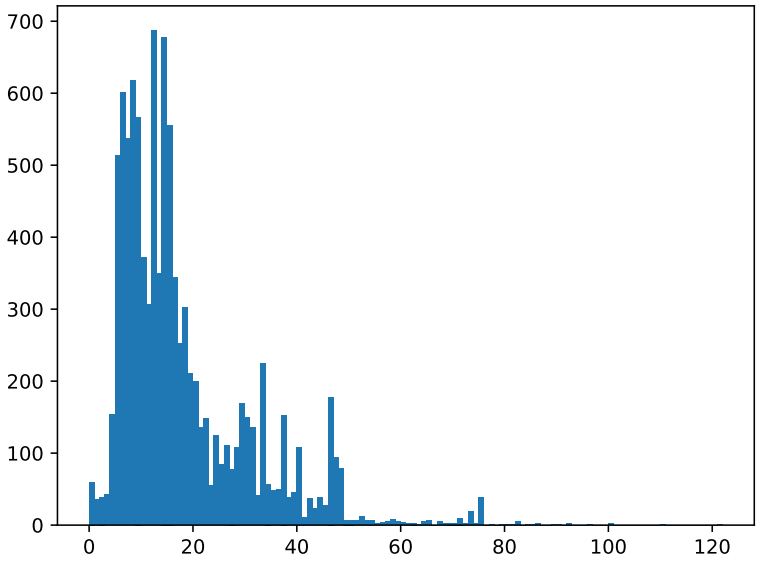


Figure 5.9: Different value set sizes distribution of coreutils MIPS binaries.

ARM, AArch64, and MIPS, where the *y*-axis measures the value set tuple size and the *x*-axis measures the percent of functions covered. Most functions have value set tuple sizes 25 or smaller (after architectural neutralization of the value sets).

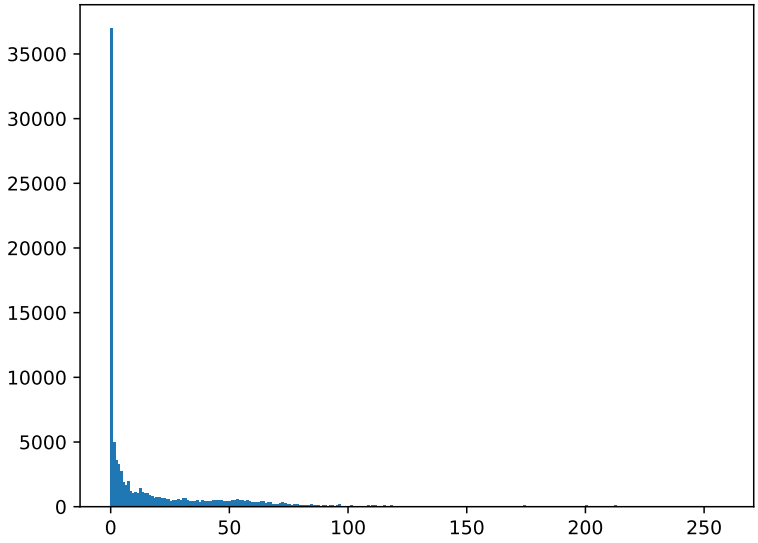


Figure 5.10: Binary code clone search ranking (starts from 0 as most similar).

5.3.2.2 Cross-architecture Evaluation

The y-axis of Figure 5.10 shows the target function ranking in our binary code search over coreutils 8.29, i.e., if the rank is 0, it means we successfully matched the binary function, and if the rank is smaller than 10, we find the target function within top 10 function candidates using value sets generated by VSA.

Table 5.4 lists ranking statics, and we observe 36.95% of functions find the correct matching in binary code search, 59.34% of functions rank the matching function within top 10 candidates, and 84.47% of functions rank the matching function within top 50 candidates.

	rank = 0	**rank < 10**	**rank < 50**
Percentage	36.95%	59.34%	84.47%

Table 5.4: The statistics of target function ranking in binary code search.

5.4 Summary

Existing binary similarity analysis approaches largely focused on extracting syntactical or structural features of the binary such as control flow graph. We observe that the value sets written to abstract locations such as registers and memory cell locations can form a unique signature and survive in syntactically different binaries. In this paper, we present a novel binary code cross-search approach named VDIFF that identifies similar functions across different architectures based on the results generated from value set analysis. We have implemented a prototype of VDIFF by performing value set analysis, architectural neutralization, and similarity detection on five architectures. The evaluation shows that VDIFF has a superior accuracy in binary code search.

Bibliography

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. Binrec: Dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[5] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2018.

[6] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[7] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization, 2017.

[8] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, page 143–157, USA, 2008. IEEE Computer Society.

[9] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[10] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[11] Sagar Chaki, Cory Cohen, and Arie Gurfinkel. Supervised learning for provenance-similarity of binaries. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 15–23, New York, NY, USA, 2011. ACM.

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 678–689, New York, NY, USA, 2016. ACM.

[13] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[14] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[15] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM.

[16] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 266–280, New York, NY, USA, 2016. ACM.

[17] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 79–94, New York, NY, USA, 2017. ACM.

[18] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 392–404, New York, NY, USA, 2018. ACM.

[19] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 349–360, New York, NY, USA, 2014. ACM.

[20] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[23] S. H. Ding, B. M. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, may 2019. IEEE Computer Society.

[24] Steven H.H. Ding, Benjamin C.M. Fung, and Philippe Charland. Kam1N0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 461–470, New York, NY, USA, 2016. ACM.

[25] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. POSTER: Rust SGX SDK: Towards memory safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2491–2493, New York, NY, USA, 2017. ACM.

[26] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium*

on Principles &Amp; Practice of Parallel Programming, PPOPP '90, pages 1–10, New York, NY, USA, 1990. ACM.

[27] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *NDSS*, 2020.

[28] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket Execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, 2014. USENIX Association.

[29] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[30] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[31] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 2016. ACM.

[32] Halvar Flake. Structural comparison of executable objects. In *DIMVA*, 2004.

[33] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[34] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pages 238–255, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. VulSeekerpro: Enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 803–808, New York, NY, USA, 2018. ACM.

[36] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, August 2017. USENIX Association.

[37] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[38] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 175–185, New York, NY, USA, 2006. ACM.

[39] hot-calls, 2017. `https://github.com/oweisse/hot-calls`.

[40] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[41] Intel. *Intel Software Guard Extensions Programming Reference*, 10 2014. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[42] Intel. *Intel Software Guard Extensions Developer Guide*, 6 2019. `https://download.01.org/intel-sgx/linux-2.6/docs/Intel_SGX_Developer_Guide.pdf`.

[43] Intel. SDK for Intel Software Guard Extensions, 2020. `https://software.intel.com/en-us/sgx/sdk`.

[44] Intel. Intel Software Guard Extensions SSL, 2021. `https://github.com/intel/intel-sgx-ssl`.

[45] intel-sgx-deep-learning, 2019. `https://github.com/landoxy/intel-sgx-deep-learning`.

[46] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 81–96, USA, 2013. USENIX Association.

[47] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX '17, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[49] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *In Proc. of the 19 th NDSS*, 2012.

[50] Y. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 756–765, May 2011.

[51] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 505–518, Berlin, Heidelberg, 2005. Springer-Verlag.

[52] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Ulf Kargén and Nahid Shahmehri. Towards robust instruction-level trace alignment of binary code. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 342–352. IEEE Press, 2017.

[54] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[55] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 971–985, New York, NY, USA, 2020. Association for Computing Machinery.

[56] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 329–338, May 2013.

[57] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In

Alfonso Valdes and Diego Zamboni, editors, *Recent Advances in Intrusion Detection*, pages 207–226, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[58] Yonghwi Kwon, Weihang Wang, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. CPR: Cross platform binary code reuse via platform independent trace program. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 158–169, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[60] 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, Vancouver, BC, August 2017. USENIX Association.

[61] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symposium*, 2013.

[62] LibSEAL, 2018. `https://github.com/lsds/LibSEAL`.

[63] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium*, NDSS'10, 2010.

[64] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, 2017. USENIX Association.

[65] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of Malicious Code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 349–358, New York, NY, USA, 2012. Association for Computing Machinery.

[66] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αDiff: Cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 667–678, New York, NY, USA, 2018. ACM.

[67] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[68] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[69] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[70] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[71] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 389–400, New York, NY, USA, 2014. ACM.

[72] Microsoft. Open Enclave SDK, 2020. `https://openenclave.io/sdk/`.

[73] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 308–319, New York, NY, USA, 2016. ACM.

[74] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, Washington, D.C., 2015. USENIX Association.

[75] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 253–270, Vancouver, BC, 2017. USENIX Association.

[76] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *19th International Conference on Cryptographic Hardware and Embedded Systems - CHES 2017*, pages 69–90, 2017.

[77] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses*, pages 423–444. Springer International Publishing, 2018.

[78] musl-libc. musl-libc, 2020. `https://www.musl-libc.org/`.

[79] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, May 2019.

[80] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[81] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[82] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ryoichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing*, pages 295–307, Boston, MA, 2005. Springer US.

[83] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[84] Andre Pawlowski, Victor van der Veen Moritz Contag, Thorsten Holz Chris Ouwehand, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[85] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, May 2015.

[86] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings*

of the 30th Annual Computer Security Applications Conference, ACSAC '14, page 406–415, New York, NY, USA, 2014. Association for Computing Machinery.

[87] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[88] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 15–27, New York, NY, USA, 2006. ACM.

[89] 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, Santa Clara, CA, August 2019. USENIX Association.

[90] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[91] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 195–209, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Nguyen Anh Quynh. Capstone: the ultimate disassembly framework. `http://www.capstone-engine.org/`, 2018.

[93] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[94] Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet. Static taint-analysis on binary executables. `http://stator.imag.fr/w/images/2/21/Laurent_Mounier_2013-01-28.pdf`, October 2011.

[95] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[96] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2018.

[97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[98] SGX_SQLite, 2018. `https://github.com/yerzhan7/SGX_SQLite`.

[99] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. A retargetable static binary translator for the arm architecture. *ACM Trans. Archit. Code Optim.*, 11(2), June 2014.

[100] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[101] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[102] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2011.

[103] Asia Slowinski, Traian Stancescu, and Herbert Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 125–137, 2012.

[104] stealthdb, 2019. `https://github.com/cryptograph/stealthdb`.

[105] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code Relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 702–714, New York, NY, USA, 2016. ACM.

[106] TaLoS, 2019. `https://github.com/lsds/TaLoS`.

[107] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[108] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[109] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 178–195, New York, NY, USA, 2018. ACM.

[110] Victor van der Veen, Dennis Andriesse, , Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *22nd ACMComputer and Communications Security CCS*, pages 927–940, 2015.

[111] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *37th IEEESecurity & Privacy S&P*, pages 934–953, 2016.

[112] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[113] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., 2015. USENIX Association.

[114] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 319–330, Piscataway, NJ, USA, 2017. IEEE Press.

[115] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 280–290, New York, NY, USA, 2009. ACM.

[116] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 440–457, Cham, 2016. Springer International Publishing.

[117] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[118] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 363–376, New York, NY, USA, 2017. ACM.

[119] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[120] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song. SPAIN: Security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472, May 2017.

[121] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[122] Fan Zhang. SGX-mbedtls, 2019. `https://github.com/bl4ck5un/mbedtls-SGX`.

[123] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 111–121, New York, NY, USA, 2012. ACM.

[124] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-program Path Sampling and Per-path Abstract Interpretation. In *Proceedings of the ACM on Programming Languages Volume 3 Issue OOPSLA (OOPSLA 2019)*, 2019.

[125] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.

[126] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.