PROTECTING COMPUTER SYSTEMS THROUGH ELIMINATING OR ANALYZING VULNERABILITIES

A Thesis
Presented to
The Academic Faculty

by

Byoungyoung Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology
August 2016

PROTECTING COMPUTER SYSTEMS THROUGH ELIMINATING OR ANALYZING VULNERABILITIES

Approved by:

Professor Wenke Lee, Co-Advisor
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Taesoo Kim, Co-Advisor
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor William R. Harris
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Alessandro Orso
School of Computer Science
College of Computing
Georgia Institute of Technology

Dr. Weidong Cui
Security and Privacy Research Group
Microsoft Research Redmond

Date Approved: 14 July 2016

Enter through the narrow gate. For wide is the gate and broad is the road that leads to destruction, and many enter through it.

Matthew 7:13-14

ACKNOWLEDGEMENTS

My Ph.D. thesis would not be possible without the support and help of a number of people. Foremost, I was very fortunate to have awesome co-advisors, Dr. Wenke Lee and Dr. Taesoo Kim. I hope one day I can be a good advisor to my future students as Wenke and Taesoo have been to me.

Since my first day at Georgia Tech, Dr. Wenke Lee has been always supportive. I remember my communication skill was shamefully bad (though I still need to improve), but he never complaint and tried to listen what I was trying to say. I am also thankful for encouraging me even during the days I lost the interests in the research. His professional vision, broader views, and critical thinking truly helped me to think like an independent researcher.

I have known Dr. Taesoo Kim for more than 10 years, and since then he has been my role model in many aspects. Not to mention his great background and deep knowledge in computer systems and security, I learned a lot from the way he thinks about the research problem—very solid logical thinking from the bottom to the top of the problem as well as trying to completely understand all the challenges and details. He has been also my great coding-teacher. It was an incredible learning experience by looking at how my code is improved by him.

I would like to thank my thesis committee members for their willingness to serve as the committee. I am thankful to Dr. Bill Harris for his creative ideas and critical comments on the research. I would like to express my gratitude to Dr. Weidong Cui, for having me as an intern at Microsoft Research where I was able to develop background in systems and security, and Dr. Alex Orso for his insightful and friendly comments during the research meeting.

This thesis is a product of great team work. I have been fortunate to collaborate with following brilliant and friendly colleagues: Dr. Simon P. Chung, Yeongjin Jang, Sanidhya Kashyap, Billy Lau, Kangjie Lu, Dr. Long Lu, Dr. Changwoo Min, Wei Meng, Chengyu Song, Tielei Wang, Dr. Xinyu Xing, and Insu Yun.

To my parents back in South Korea, I always thank you for your unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

There have been tremendous efforts to build fully secure computer systems, but it is not an easy goal. Making a simple mistake introduces a vulnerability, which can critically endanger a whole system's security.

This thesis aims at protecting computer systems from vulnerabilities. We take two complementary approaches in achieving this goal, eliminating or analyzing vulnerabilities. In the vulnerability elimination approach, we eliminate a certain class of memory corruption vulnerabilities to completely close attack vectors from such vulnerabilities. In particular, we develop tools DANGNULL and CAVER, each of which eliminates popular and emerging vulnerabilities, use-after-free and bad-casting, respectively. DANGNULL relies on the key observation that the root cause of use-after-free is that pointers are not nullified after the target object is freed. Thus, DangNull instruments a program to trace the object's relationships via pointers and automatically nullifies all pointers when the target object is freed. Similarly, CAVER relies on the key observation that the root cause of bad-casting is that casting operations are not properly verified. Thus, CaVer uses a new runtime type tracing mechanism to overcome the limitation of existing approaches, and performs efficient verification on all type casting operations dynamically. We have implemented these protection solutions and successfully applied them to Chrome and Firefox browsers. Our evaluation showed that DangNull and CaVer imposes 29% and 7.6% benchmark overheads in Chrome, respectively. We have also tested seven use-after-free and five bad-casting exploits in Chrome, and DangNull and CaVer safely prevented them all.

In the vulnerability analysis approach, we focus on a timing-channel vulnerability which allows an attacker to learn information about program's sensitive data without causing a program to perform unsafe operations. It is challenging to test and further confirm the timing-channel vulnerability as it typically involves complex algorithmic operations. We implemented SideFinder, an assistant tool identifying timing-channel vulnerabilities in a hash table. Empowered with symbolic execution techniques, SideFinder semi-automatically synthesizes inputs attacking timing-channels, and thus confirms the vulnerability. Using SideFinder, we analyzed and further synthesized two real-world attacks in the Linux kernel, and showed it can break one important security mechanism, Address Space Layout Randomization (ASLR).

CHAPTER I

INTRODUCTION

1.1 Thesis Overview

Today's computer systems are large and complex. There have been tremendous research efforts to fully defend these systems, but it is still an elusive goal. Introducing radical design changes for security is not practical, or adding a small security enhancing component easily breaks the critical functionality of a system. Even if these issues are resolved, attackers are ever evolving, and they always find a vulnerability from tiny little corner cases.

This thesis aims at protecting computer systems through two complementary approaches, eliminating or analyzing vulnerabilities. Eliminating critical vulnerabilities provides strong notion of system protections. Unlike preventing certain side-effects caused by vulnerabilities, this elimination approach directly fixes a root cause of vulnerabilities, thereby leaving no security holes that attackers may attempt to abuse in the future. In particular, this thesis focuses on eliminating two popular and emerging vulnerabilities, use-after-free and bad-casting, each of which is addressed by DANGNULL and CAVER, respectively. DANGNULL and CAVER have been implemented and then applied to popular web browsers such as Chrome and Firefox, and demonstrated its effectiveness in terms of both performances and security. More importantly, these tools helped to identify eleven new security vulnerabilities in Firefox and `GNU Libc++`, all of which have been fixed by respected vendors.

In addition, analyzing vulnerabilities is an important asset to protect computer systems. In particular, this thesis develops a tool, SIDEFINDER, which helps to unveil a timing-channel vulnerability in hash tables. While timing-channels can be critical security issues if abused by an attacker to leak security sensitive data, but it is challenging to identify such vulnerabilities. This is largely because suspecting a program/algorithm to be vulnerable to timing-channel may be easy, but generating concrete input to verify such a hypothesis is difficult. Using SIDEFINDER, this thesis analyzes two new instances of such vulnerabilities in the Linux kernel, and show these problems can break traditional security issues including Address Space Layout Randomization (ASLR).

The following three subsections introduce the problem scope and research approaches taken in this thesis. Then we summarize the contribution of this thesis.

1.2 Eliminating Use-after-free Vulnerability

Use-after-free remains one of the most critical and popular attack vectors because existing proposals have not adequately addressed the challenging program analysis and runtime performance issues. DANGNULL (§2) is a system that detects temporal memory safety violations—in particular, use-after-free and double-free—during runtime. DANGNULL relies on the key observation that the root cause of these violations is that pointers are not nullified after the target object is freed. Based on this observation, DANGNULL automatically traces the object's relationships via pointers and automatically nullifies all pointers when the target object is freed. DANGNULL offers several benefits. First, DANGNULL addresses the root cause of temporal memory safety violations. It does not rely on the side effects of violations, which can vary and may be masked by attacks. Thus, DANGNULL is effective against even the most sophisticated exploitation techniques. Second, DANGNULL checks object relationship information using runtime object range analysis on pointers, and thus is able to keep track of pointer semantics more robustly even in complex and large scale software. Lastly, DANGNULL does not require numerous explicit sanity checks on memory accesses because it can detect a violation with implicit exception handling, and thus its detection capabilities only incur moderate performance overhead.

1.3 Eliminating Bad-casting Vulnerability

Many applications such as the Chrome and Firefox browsers are largely implemented in C++ for its performance and modularity. Type casting, which converts one type of an object to another, plays an essential role in enabling polymorphism in C++ because it allows a program to utilize certain general or specific implementations in the class hierarchies. However, if not correctly used, it may return unsafe and incorrectly casted values, leading to so-called *bad-casting* or *type-confusion* vulnerabilities. Since a bad-casted pointer violates a programmer's intended pointer semantics and enables an attacker to corrupt memory, bad-casting has critical security implications similar to those of other memory corruption vulnerabilities. Despite the increasing number of bad-casting vulnerabilities, the bad-casting detection problem has not been addressed by the security community.

CAVER (§3) is a runtime bad-casting detection tool. It performs program instrumentation at compile time and uses a new runtime type tracing mechanism—the type hierarchy table—to overcome the limitation of existing approaches and efficiently verify type casting dynamically. In particular, CAVER can be easily and automatically adopted to target applications, achieves broader detection coverage, and incurs reasonable runtime overhead.

1.4 Analyzing Hash Table Timing-channel Vulnerability

Timing-channel vulnerabilities allow an attacker to learn information about program's sensitive data without causing a program to perform unsafe operations. We particularly focus on a class of timing-channel vulnerabilities in programs that use deterministic hash tables to store sensitive data as keys. It is challenging to test and further confirm such vulnerabilities because it requires heavy manual reverse engineering efforts given the complexity of underlying software.

We develop SIDEFINDER (§4), which automatically synthesizes inputs attacking timing-channel vulnerabilities in hash tables. Using SIDEFINDER, we analyzed two new instances of such vulnerabilities in the Linux kernel that allows an attacker to learn the values of private data or addresses of key system objects, in some cases breaking Address Space Layout Randomization (ASLR), and show these problems can be reduced to SIDEFINDER. We implemented and evaluated SIDEFINDER for timing-channel vulnerabilities in the Linux Kernel, and confirmed that SIDEFINDER is able to synthesize inputs attacking timing-channels.

1.5 Thesis Contributions

In summary, this thesis makes the following technical contributions.

- **Analysis and formalization of vulnerabilities**: This thesis presents the analysis and formalization on emerging vulnerabilities, namely use-after-free, bad-casting, and side-channel timing attacks. The analysis includes details on its security impacts in details so that security researchers and practitioners can better understand these vulnerabilities.
- **Practical security enhancing tools**: This thesis develops three tools eliminating or analyzing vulnerabilities. We have thoroughly evaluated its performance aspects as well as security effectiveness in the commodity software (i.e., Chrome, Firefox, and the Linux kernel) to maximize its practical impacts.
- **New vulnerabilities**: This thesis discovered fourteen previously unknown vulnerabilities in the commodity software. Most of vulnerabilities have been reported to the corresponding vendors, which helped them to fix and secure their products.

CHAPTER II

DANGNULL:NULLIFYING DANGLING POINTERS TO PREVENT USE-AFTER-FREE

2.1 Eliminating Use-after-free Vulnerability

Many system components and network applications are written in the unsafe C/C++ languages that are prone to memory corruption vulnerabilities. To address this problem, a large number of techniques have been developed to improve memory safety and prevent memory corruption bugs from being exploited [3, 12, 43, 55, 87, 88, 91, 92, 101, 124]. However, the problem of detecting and preventing use-after-free bugs remains unsolved. Among the CVE identifiers of the Chromium browser that we collected from Oct. 2011 to Oct. 2013 in Table 1, use-after-free vulnerabilities are not only 40x/3x more than stack and heap overflows in quantity, but also have more severe security impacts than traditional vulnerabilities: 88% of use-after-free bugs are rated critical or high in severity, while only 51% of heap overflows are considered as high severity. Not only are there many use-after-free vulnerabilities, they have also become a significant attack vector. In Pwn2Own 2014 [58], an annual contest among hackers and security research groups, the VUPEN team was awarded with the largest cash amount, $100,000, for a single use-after-free exploit that affects all major WebKit-based browsers.

Compared with many other vulnerability types, including stack buffer overflows or heap buffer overflows, use-after-free is generally known as one of the most difficult vulnerability type to identify using static analysis. In modern C/C++ applications (especially under object-oriented or event-driven designs), the resource free (i.e., memory deallocation) and use (i.e., memory dereference) are well separated and heavily complicated. Statically identifying use-after-free vulnerabilities under this difficult conditions involves solving challenging

Table 1: The number of security vulnerabilities in the Chromium browser for two years (2011–2013), classified by types of vulnerabilities and their severity.

Severity	**Use-after-free**	**Stack overflow**	**Heap overflow**	**Others**
Critical	13	0	0	0
High	582	12	107	11
Medium	80	5	98	12
Low	5	0	3	1
Total	680	17	208	24

static analysis problems (e.g., inter-procedural and point-to analysis while also considering multi-threading effects), and is therefore feasible only for small size programs [45, 94].

Most research efforts to detect use-after-free vulnerabilities are relying on either additional runtime checks or dynamic analysis (listed in Table 2). For instance, use-after-free detectors including [87, 124] have been proposed to address dangling pointer issues. By maintaining metadata for each pointer and tracking precise pointer semantics (i.e., which pointer points to which memory region), these tools can identify dangling pointers or prevent memory accesses through dangling pointers. However, precisely tracking runtime semantics on a per-pointer bases is non-trivial as there would be a huge number of pointers and their metadata in runtime, which may result in high false positive rates (i.e., identifying benign program behavior as use-after-free) or significant performance degradation. Such shortcomings would limit the potential for these techniques to be deployed for large scale software.

Memory error detectors [55, 91, 101] are also able to capture use-after-free bugs during the software testing phase. By maintaining the allocated/freed status of memory, these tools can prevent accesses to freed memory. However, these tools are not suitable for detecting real-world exploits against use-after-free vulnerabilities if attackers can partially control the heap memory allocation process, especially for web browsers. For example, by using Heap Spraying [39, 97] or Heap Fengshui [105] like techniques, attackers can force the target program to reuse certain freed memory.

In addition, Control Flow Integrity (CFI) tools can be used to prevent use-after-free

Table 2: Comparing the proposed system DangNull with other protection techniques detecting use-after-free. The `Explicit checks` column represents whether the technique explicitly instruments checks to detect use-after-free except via pointer propagation. The `Liveness support` column represents whether the technique may continuously run an instrumented application as if use-after-free vulnerabilities are patched. The `False positive rates` column represents whether the technique would generate the high/low number of false alarms on benign inputs, and high false positive rates imply that it would be difficult to be deployed for large scale software. The `Bypassable` column shows whether the protection technique can be bypassable with the following column's exploitation technique. Overall, while all other protection techniques show either high false positive rates or being bypassable, DangNull achieves both low false positive rates and being non-bypassable against sophisticated exploitation techniques. §2.7 describes more details on each protection technique.

Category	**Protection Technique**	**Explicit Checks**	**Liveness Support**	**False positive Rates**	**Bypassable**	**Bypassing Technique**
Use-after-free detectors	DangNull	No	Yes	Low	No	N/A
	CETS [87]	Yes	No	High	No	N/A
	Undangle [18]	Yes	No	Low	No	N/A
	Xu et al. [124]	Yes	No	Low	No	N/A
Memory error detectors	AddressSanitizer [101]	Yes	No	Low	Yes	Heap manipulation
	Memcheck [91]	Yes	No	High	No	N/A
	Purify [55]	Yes	No	High	No	N/A
Control Flow Integrity	CCFIR [126]	Yes	No	Low	Yes	Corrupting non-function pointers
	bin-CFI [129]	Yes	No	Low	Yes	
	SafeDispatch [61]	Yes	No	Low	Yes	
Safe memory allocators	Cling [3]	No	No	Low	Yes	Heap manipulation
	DieHarder [92]	No	No	Low	Yes	Heap manipulation

vulnerabilities from being exploited to hijack the control-flow because the majority of vulnerability exploitations hijack the control flow to execute malicious code with Return-Oriented Programming (ROP) [95]. However, due to the inherent limitations of these tools, most of them only enforce coarse-grained CFI, which leaves some control-flows exploitable [21, 40, 51, 52]. Moreover, since control-flow hijacks are not the only method to compromise a program, it is still possible to bypass these techniques even if they can enforce perfect CFI, e.g., via non-control data attacks [25, 80].

Overall, all of the previous protection techniques show either high false positive rates or are bypassable using certain exploitation techniques. In other words, there is currently no use-after-free mitigation solution that works well for large scale software and can also stop all known forms of use-after-free exploitation techniques.

In this regard, we present DangNull, a system that prevents temporal memory safety violations (i.e., use-after-free and double-free) at runtime. As suggested by many secure programming books [99], a pointer should be set to NULL after the target object is freed. Motivated by the fact that dangling pointers obviously violate this rule, DANGNULL automatically traces the object relationships and nullifies their pointers when the object they pointed to is freed. In particular, rather than relying on a heavy dynamic taint analysis, DANGNULL incorporates a runtime object range analysis on pointers to efficiently keep track of both pointer semantics and object relationships. Based on the collected object relationship information, DANGNULL nullifies dangling pointers when the target memory is freed. After this nullification, any temporal memory safety violation (i.e., dereferencing the dangling pointers) turns into a null-dereference that can be safely contained.

This unique design choice of DANGNULL offers several benefits. First, since nullification immediately eliminates any possible negative security impacts at the moment dangling pointers are created, DANGNULL does not rely on the side effects from use-after-free or double-free, and thus cannot be bypassed by sophisticated exploit techniques. Second, a runtime object range analysis on pointers allows DANGNULL to efficiently keep track of pointer semantics. Instead of tracing complicated full pointer semantics, DANGNULL only tracks abstracted pointer semantics sufficient to identify dangling pointers with the understandings on runtime object ranges. This allows DANGNULL to overcome the difficulty of pointer semantic tracking and scale even for complex and large software. Third, DANGNULL does not require any explicit sanity checks on memory accesses, which are a common performance bottleneck in other mitigation tools. Instead, it relies on implicit null-dereference exceptions, which are safely contained by DANGNULL. Lastly, DANGNULL can continue running correctly even after use-after-free exploits (i.e., as if a program is patched for the exploiting vulnerability) in some cases. Since nullified dangling pointers have identical semantics with existing null-pointer checks in programs, it can utilize such existing checks and survive use-after-free exploits.

We implemented DANGNULL and applied it to SPEC CPU 2006 and Chromium, and evaluated its effectiveness and performance overhead. In particular, for the Chromium browser, all seven real-world use-after-free exploits we tested were safely prevented with DANGNULL. Moreover, DANGNULL only imposed an average of 4.8% increase in overhead in JavaScript benchmarks and an average of 53.1% increase in overhead in rendering benchmarks, while a page loading time on Alexa top 100 websites showed a 7.4% slowdown on average. DANGNULL also showed no false positives while running more than 30,000 benign test cases (including both unit tests and end-to-end testing web pages).

To summarize, this chapter makes the following contributions:

- We proposed DANGNULL, a system that detects temporal memory safety violations, and is resilient to sophisticated bypassing techniques.
- We implemented DANGNULL and apply it to the Chromium browser, a well known complex and large scale software.
- We thoroughly evaluated various aspects of DANGNULL: attack mitigation accuracy, runtime performance overheads, and compatibility.

2.2 Background

From dangling pointers to use-after-free. Dangling pointers refer to pointers that point to freed memory, and lead to memory safety errors when accessed. To be precise, a dangling pointer itself does not cause any memory safety problem, but accessing memory through a dangling pointer can lead to unsafe program behaviors and even security compromises, such as control-flow hijacking or information leakage.

For example, as illustrated in Figure 1, `body` and `doc`→`child` pointers become dangling pointers after `body` is deleted; the object `body` is allocated, assigned to `doc`→`child`, and freed after the propagation. Since `doc`→`child` now points to the invalid memory region (which is freed or already reused by the memory allocator), if the memory is accessed by `doc`→`child`, a use-after-free error occurs. Unfortunately, use-after-free errors often lead to security exploits. For example, if an adversary can manipulate (directly or probabilistically) the memory region, he can then use the pointer to obtain sensitive data or even change the program's control flows.

Challenges in identifying dangling pointers. The example we described above is very simple. In practice, however, real-world use-after-free vulnerabilities can be complicated. First, the allocation, propagation, free, and dereference operations could all be located in separate functions and modules. Second, at runtime, the execution of these operations could occur in different threads. Especially for applications with event-driven designs and object-oriented programming paradigms in mind, the situation becomes even worse. For example, web browsers need to handle various events from JavaScript or DOM, UI applications need to handle user generated events, and server side applications implement event-loops to handle massive client requests. Given the complicated and disconnected component relationships, developers are prone to make mistakes when implementing object pointer operations and thus leave the door open for dangling pointer problems.

In addition, not all dangling pointers violate temporal memory safety. In our experiments (§2.5), we found numerous benign dangling pointers in practice (e.g., there were between 7,000 and 32,000 benign dangling pointers while visiting benign web pages in the Chromium browser). Thus, in order to clarify dangling pointer problems, we first define the following notions:

Definition 1. Dangling pointer. *A pointer variable p is a dangling pointer if and only if*

$$\begin{aligned}(x := allocate(size)) \wedge p \in [x, x + size - 1] \\ \vdash \ deallocate(x)\end{aligned}$$

for any pointer variables x and p, and a value size.

This definition simply captures the fact that a dangling pointer points to a freed memory area; if the variable p points to the freed memory area ($[x, x + size - 1]$), p is a dangling pointer. Without loss of generality, we assume that `allocate()` and `deallocate()` functions denote all memory allocation and deallocation functions, respectively. Based on this definition, we further define unsafe and benign dangling pointers.

Definition 2. Unsafe dangling pointer. *A pointer variable p is an unsafe dangling pointer if p is a dangling pointer and there exists memory read or write dereferences on p*

Definition 3. Benign dangling pointer. *A pointer variable* p *is a benign dangling pointer if* p *is a dangling pointer but not an unsafe dangling pointer.*

Note that only unsafe dangling pointers violate temporal memory safety and should thus be prevented or detected. On the other hand, an alarm on a benign dangling pointer is considered a false alarm (i.e., false positive).

Exploiting dangling pointers. In order to abuse unsafe dangling pointers for security compromises (e.g., to achieve control flow hijacking or information leak), an attacker needs to place useful data in the freed memory region where an unsafe dangling pointer points. Depending on how unsafe dangling pointers are subsequently used, different exploitation techniques are employed. For example, attackers can place a crafted virtual function table pointer in the freed region; and when a virtual function in the table is invoked later (i.e., memory read dereference on unsafe dangling pointers), a control flow hijacking attack is accomplished. As another example, if the attacker places a root-privileged flag for checking the access rights in the freed region, a privilege escalation attack is accomplished. Moreover, if the attacker places corrupted string length metadata in the freed region and the corresponding string is retrieved, an information leakage attack is accomplished.

It should be clear now that the exploitability of unsafe dangling pointers depends on whether an attacker can place crafted data where a dangling pointer points. Specifically, an attacker needs to place useful objects where the freed memory region is located (e.g., an extra memory allocation is one popular exploitation technique). This implies that the extra operations controlled by an attacker should be performed between free and use operations (e.g., between line 17 and 21 in Figure 1) because it is the only time window that the freed memory region can be overwritten. In this sense, security professionals determine the exploitability of use-after-free or double-free based on whether an attacker can gain control between the free and use operations. For example, the Chromium security team uses this notion to determine the amount of bug bounty rewards [112].

2.3 Design

DANGNULL aims to detect unsafe dangling pointers without false positives and false negatives in practice, so that it cannot be bypassed with sophisticated exploitation techniques while supporting large scale software. To achieve this goal, DANGNULL addresses the root cause of the unsafe dangling pointer problem. As discussed in §2.2, the root cause of unsafe dangling pointers is that pointers—including both 1) the pointer that an allocated memory object is initially assigned to and 2) all pointers that are propagated from the initial pointer through direct copy and pointer arithmetic —are not nullified after the target memory is freed.

Based on this observation, DANGNULL is designed to 1) automatically trace the point-to relations between pointers and memory objects, and 2) nullify all pointers that point to a freed memory object. By nullifying pointers that would otherwise have become unsafe dangling pointers, DANGNULL not only prevents reads and writes of the freed memory, which may contain security-sensitive meta-data (e.g, function pointers or vector length variables), but also, in many cases, shepherds the execution of applications as if the detected use-after-free or double-free bug had already been patched.

2.3.1 System Overview

An overview of DANGNULL's design is illustrated in Figure 2. To generate a binary secured against use-after-free vulnerabilities, developers should compile the source code of the target program with DANGNULL. Given the source code, DANGNULL first identifies instructions that involve pointer assignments and then inserts a call to the tracing routine (a static instrumentation in §2.3.2). At runtime, with the help of instrumented trace instructions, DANGNULL keeps track of point-to relations in a thread-safe red-black tree, `shadowObjTree`, that efficiently maintains shadow objects representing the state of corresponding memory objects (a runtime library in §2.3.3).

On every memory allocation, DANGNULL initializes a shadow object for the target object being created. Upon freeing an object, DANGNULL retrieves all pointers that point to this object (from `shadowObjTree`) and nullifies those pointers, to prevent potential use-after-free or double-free.

Later in this section, we describe each component of DANGNULL (the static instrumentation and the runtime library), and explain how we maintain `shadowObjTree` with a concrete running example (Figure 1).

2.3.2 Static Instrumentation

The static instrumentation of DANGNULL is done at the LLVM IR [75] level and is designed to achieve one primary goal: to monitor pointer assignments to maintain the point-to relations. To balance security, performance, and engineering efforts, only appropriate pointers are instrumented. More specifically, DANGNULL only tracks pointers located on the heap (e.g., `doc`→`child` in Figure 1) but not on the stack (e.g., `doc` in Figure 1). From our preliminary experiment on the Chromium browser, we found that stack-located pointers are unlikely to be exploitable, even though are many dangling pointers. This is because stack-located pointers tend to have a very short lifetime since the scope of stack variables are bounded by the scope of a function and accesses to those variables are limited in the programming language. Heap-located pointers generally have much a longer lifetime (i.e., the number of instructions between free and use is larger). In other words, unsafe dangling pointers located in the heap offer better controls between the free and dereference operations, and are thus are more likely to be exploited (§2.2). Therefore, to reduce performance overhead and keep our engineering efforts effective and moderate, we focus on heap-located pointers. Note that the nullification idea of DANGNULL has no dependencies on the pointer locations, and is generally applicable to both heap- and stack-located pointers.

The algorithm for the static instrumentation is described in Figure 3. At lines 1-4, all store instructions [1] in each function are iterated. With the pointer information obtained at lines 5-6, DANGNULL first opts out if lhs is a stack variable, using a intra-procedure backward data-flow analysis (line 9-10). Specifically, given a lhs variable, we leveraged a def-use chain provided by LLVM to see if this variable is allocated on the stack via the allocation statement. Since this analysis is conservative, it is possible that DANGNULL still instruments some pointer assignments in the stack. However, as DANGNULL does not instrument allocations and deallocations of stack variables, such assignments will be ignored by the runtime library. Next, DANGNULL performs a simple type signature check to see if rhs is not of a pointer type (line 11-12) [2]. With these two opt-out checks, DANGNULL ignores all uninteresting cases as the current version of DANGNULL only targets the heap located pointers. Because the location of a heap pointer cannot always be statically known due to pointer aliasing issues, store instructions are conservatively instrumented unless it is soundly determined to be a stack-located pointer. Any possible over-instrumentation due to this conservative choice will be handled using the runtime object range analysis, which we will describe in the next subsection (§2.3.3).

Once all these sanity checks are passed, a `trace()` function call is inserted after the store instruction. For example, to instrument `doc->child = body` in Figure 1, DANGNULL inserts `trace(&doc->child, body)` after its assignment instruction. In this way, the DANGNULL's runtime library can later correctly trace the pointer references originating from `doc->child`.

Note that DANGNULL relies on the type signature of C/C++. Coercing type signatures in the code might cause some false negatives, meaning that DANGNULL can miss some propagation of pointers at runtime. In particular, if developers convert types of pointer objects (by casting) into a value of non-pointer types, then DANGNULL will not be able to trace the pointer propagation via that value. Moreover, if some libraries are not built using DANGNULL (e.g., proprietary libraries), DANGNULL would still be able to run them

[1] In the LLVM IR, store instructions are always in the form of `lhs := rhs`.

[2] In the LLVM IR, the type of `lhs` of a store instruction is always the pointer of the `rhs`'s type.

together, but the protection domain will be limited only to the instrumented code or modules.

2.3.3 Runtime Library

The runtime library of DANGNULL maintains all the object relationship information with an efficient variant of a red-black tree, called `shadowObjTree`. Object layout information (i.e., address ranges of an object) is populated by interposing all memory allocation and deallocation functions (e.g., `malloc` and `free`, `new` and `delete`, etc). Object relationships (i.e., an object refers to another object) are captured with the help of `trace()` added during the static instrumentation. Based on the collected object relationship information, the runtime library automatically nullifies all dangling pointers when the target memory is freed.

In this subsection, we first describe `shadowObjTree`, a data structure designed for maintaining the complex object relationships (§2.3.3.1). We then further describe how these data structures are populated and how dangling pointers are immediately nullified during runtime (§2.3.3.2).

2.3.3.1 Shadow Object Tree

DANGNULL records and maintains the relationships between objects[3] in `shadowObjTree`. It has a hierarchical structure because the object relationship itself is hierarchical: each running process has multiple objects, and each object has multiple in/out-bound pointers. Thus, `shadowObjTree` is composed of several sub-data structures as nodes, to better represent this hierarchical structure.

Figure 4 shows a structural view of `shadowObjTree`. A node of `shadowObjTree` is a shadow object, which holds the object's memory layout information (i.e., the object boundary with base and end addresses) and in/out-bound pointers (i.e., directed references between objects). To find a shadow object for the given pointer p, `shadowObjTree` searches for a shadow object such that the corresponding object's $base \leq p < end$. In other words, as long

[3] Since DANGNULL only tracks pointers stored on heap, the point-to relationship effectively becomes a relationship between heap objects.

as the pointer p points to a corresponding object's address range, `shadowObjTree` returns the shadow object.

To efficiently support operations like insert, remove, and search, `shadowObjTree` uses a variant of the red-black tree as the underlying data structure, which generally excels if 1) the insertion order of keys shows random patterns and 2) the number of search operations is significantly more than that of insertion and remove operations. In `shadowObjTree`, the key is the address of the object, and the order of allocated objects' addresses eventually depends on the `mmap()` system call, which shows random behavior in modern ASLR enabled systems. Moreover, DANGNULL requires significantly more `find()` operations than `allocObj()` and `freeObj()` operations to soundly trace the object relationships.

Note that a hash table would not be an appropriate data structure for `shadowObjTree` because it cannot efficiently handle the size information of an object. To be specific, a find operation of `shadowObjTree` must answer range-based queries (i.e., finding a corresponding shadow object given the address, where the address can point to the middle of a shadow object), but a hash function of a hash table cannot be efficiently designed to incorporate such range information.

In addition, `shadowObjTree` has two sub-trees to maintain in/out-bound pointer information, and each sub-tree uses a red-black tree as the underlying data structure for the same reason described for `shadowObjTree`. As the pointer reference is directed, an in-bound reference of the object denotes that the object is pointed to by some other object and an out-bound reference denotes that it points to some other object. For example, the `body` object in Figure 1 has `doc`→`child` as an in-bound sub-tree and `div` as an out-bound sub-tree.

2.3.3.2 Runtime Operations and Nullification

Upon running the instrumented binary, the runtime library of DANGNULL interposes all memory allocations and deallocations, and redirects their invocations to `allocObj()` and freeObj(). In addition, trace() instructions were inserted at pointer propagation instructions from the static instrumentation. As a running example, Figure 5 illustrates how DANGNULL interposes and instruments the example code in Figure 1 where + marked lines show the interposed or instrumented code.

The algorithm for the runtime library, which populates shadowObjTree, is described in Figure 6. Upon the memory allocation invocation, allocObj() first invokes corresponding real allocation functions (line 2). With the base pointer address from the real allocation, a shadow object is created and inserted to shadowObjTree as a node (lines 3-4). When trace() is invoked, the object relationship is added to the shadow objects. It first fetches two shadow objects representing lhs and rhs pointers, respectively (line 9-10). Next, with the concrete runtime values on pointers, DANGNULL uses the object range analysis to check whether lhs and rhs truly point to live heap objects (line 13). It is worth noting that this object range analysis not only helps DANGNULL avoid tracing any incorrect or unnecessary pointer semantics that are not related to dangling pointer issues, but also makes DANGNULL more robust on object relationship tracings, since it is based on concrete values and reasons about the correctness of pointer semantics with the liveness of source and destination heap objects. If the check passes, DANGNULL first removes an existing relationship, if there is any (line 14). It then inserts the shadow pointer to both shadow objects (line 16-17).

Note, by using shadowObjTree, DANGNULL does not need to handle pointer arithmetic to trace pointers. Specifically, because shadowObjTree contains information (base and size) of all live heap objects, given any pointer p, DANGNULL can locate the corresponding object through a search query (shadowObjTree.find()), i.e., finding object that has its $\texttt{base} \leq q < (\texttt{base + size})$. For the same reason, although DANGNULL does not trace non-heap-located pointers (i.e., pointers in the stack or global variables), DANGNULL can still trace correctly when the pointer value is copied through them and brought back to the heap.

When an object is freed with freeObj(), the actual nullification starts. DANGNULL first fetches its shadow object (line 21). Next, it iterates over all in-bound pointers (pointing to the current object to be freed), and nullifies them with a pre-defined value (`NULLIFY_VALUE` at line 27). Note that these in-bound pointers are the pointers that would become dangling otherwise, and the pre-defined value can be set as any relatively small non-negative integer value (e.g., 0, 1, 2, ...). To avoid the erroneous nullification due to later deallocation of objects that the current object points to, DANGNULL also removes the current object from the sub-tree of the out-bound pointers (lines 29-31).

It is worth noting that DANGNULL nullifies not only unsafe dangling pointers, but also benign dangling pointers. In spite of this extra nullification, DANGNULL can still retain the same program behavior semantics because benign dangling pointers should not have any pointer semantics (i.e., never be used). In most cases as we quantified in §2.5, DANGNULL behaves correctly without false positives. We have found one rare false positive case, described in detail in §2.6.

In our secured binary (Figure 5), `doc`→`child` is automatically nullified when `body` is freed: the shadow object representing `body` was created (line 3), propagated to `doc`→`child` (line 8), and nullified when the `body` is deallocated (line 15). As a result, depending on `NULLIFY_VALUE`, the example would raise the `SIGSEGV` exception (if `NULLIFY_VALUE` > 0) or continuously run (if `NULLIFY_VALUE` == 0), both of which safely mitigates negative security impacts by unsafe dangling pointers.

For the `SIGSEGV` exception cases, DANGNULL guarantees that the program securely ends in a safe-dereference, which is defined as follows.

Definition 4. Safe-dereference. *If a dereference instruction accesses the memory address in the range of [0, N] where it is preoccupied as non-readable and non-writable memory pages for a given constant N, such a dereference is a safe-dereference.*

A safe-dereference guarantees that a dereference on nullified unsafe dangling pointers turns into a secured crash handled either by the operating system or DANGNULL's `SIGSEGV` exception handler. In modern operating systems, it is common that the first several virtual memory pages are protected to avoid any potential null-dereference attacks (e.g., virtual address spaces from 0 to 64K are protected in Ubuntu [120]). In other words, DANGNULL can utilize this existing null address padding to guarantee safe-dereferences (64K in Ubuntu). Even if this null address padding is not supported by the operating system, DANGNULL can still pre-allocate these spaces using the `mmap()` system call to be non-readable and non-writable before any other code runs.

For continuously running cases, DANGNULL utilized the existing sanity check at line 18. This is because the semantic on invalid pointers is identical to both DANGNULL's nullification and typical programming practices. In other words, because it is common for developers to check whether the pointer value is null before accessing it, DANGNULL's nullification can utilize such existing checks and keep the application running as if there were no unsafe dangling pointers.

This example is oversimplified for the purpose of clarifying the problem scope and showing how DANGNULL can nullify dangling pointers. In §2.5.1, we show that DANGNULL is effective when applied to real, complex use-after-free bugs in Chromium.

2.4 Implementation

We implemented DANGNULL based on the LLVM Compiler project [114]. The static instrumentation module is implemented as an extra LLVM pass, and the runtime library is implemented based on LLVM compiler-rt with the LLVM Sanitizer code base. Table 3 shows the lines of code to implement DANGNULL, excluding empty lines and comments.

We placed the initialization function into `.preinit_array` as a `ELF` file format so that the initialization of DANGNULL is done before any other function[4]. In this function, all standard allocation and deallocation functions (e.g., `malloc` and `free`, `new` and `delete`, etc) are interposed. In total, DANGNULL interposed 18 different allocation functions in the current implementation, and any additional customized allocators for the target application

[4] DANGNULL's prototype is implemented on a Linux platform. Although several implementation details are specific to Linux, these can be generally handled in other platforms as well.

Table 3: Components of DANGNULL and their complexities, in terms of their lines of code. All components are written in C++.

Components	**Lines of code**
Static Instrumentation	389
Runtime Library	3,955
`shadowObjTree`	1,303
Red-black tree	476
Runtime function redirection	233
Others	1,943
Total	4,344

can be easily added with one line of its function signature.

To avoid multi-threading issues when running DANGNULL, we used mutex locks for any data structures with the potential for data racing. One global mutex lock is used for `shadowObjTree`, and all shadow objects and their in/out–bound pointer sub-trees also hold their own mutex locks.

To retain the target program's original memory layout, DANGNULL uses a dedicated allocator from Sanitizer that has dedicated memory pages. All memory for metadata, including `shadowObjTree` and its pointer sub-trees, is allocated from this allocator. Thus, DANGNULL does not interfere with the original memory layout, and it can avoid any potential side effects by manipulating the original allocators [46].

We also modified the front-end of LLVM so that users of DANGNULL can easily build and secure their target applications with one extra compilation option and linker option. To build SPEC CPU 2006 benchmarks, we added one line to the build configuration file. To build the Chromium browser, we added 21 lines to the `.gyp` build configuration files.

2.5 Evaluation

We evaluated DANGNULL on two program sets, the SPEC CPU2006 benchmarks [107] and the Chromium browser [110] [5]. First, we tested how accurately DANGNULL mitigates known use-after-free exploits (§2.5.1). Next, we measured how much overhead DANGNULL

[5] The Chromium browser is the open source project behind the Chrome browser, and these two are largely identical.

imposes during the instrumentation phase (§2.5.2) and the runtime phase (§2.5.3). Finally, we conducted a stress test to see if DANGNULL runs well without breaking compatibility (§2.5.4). All experiments were conducted on an Ubuntu 13.10 system (Linux Kernel 3.11.0) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

2.5.1 Attack Mitigation

The goal of DANGNULL is to turn use-after-free or double-free attempts into safe-dereferences by nullifying dangling pointers. In order to test how DANGNULL accurately nullified unsafe dangling pointers and eventually protected the system from temporal memory safety violations, we tested the DANGNULL-hardened Chromium browser with real-world use-after-free exploits. Given the Chromium version instrumented (29.0.1457.65), we first collected all publicly available use-after-free exploits from the Chromium bug tracking system [112], which opens vulnerability information to the public after mitigation and includes a proof of concept exploit[6].

Table 4 lists seven use-after-free vulnerabilities that existed in the targeted Chromium version. All of these were marked as high severity vulnerabilities by the Chromium team, which suggests that these have a high potential to be used for arbitrary code execution. Bug ID 162835 was specifically selected to later demonstrate that DANGNULL can mitigate this sophisticated exploit technique.

Before applying DANGNULL, all proofs-of-concept can trigger `SIGSEGV` exceptions at invalid addresses (`No-Nullify` column in Table 4). These invalid addresses are memory addresses that are dereferenced, i.e., the values of unsafe dangling pointers. Although we only present one value for each vulnerability, this value would randomly change between different executions due to ASLR and the order of memory allocations. These seemingly random

[6] We have not found any double-free vulnerabilities for the given Chromium version. However, we believe DANGNULL would be equally effective against double-free exploits because DANGNULL nullifies exploit attempts where both use-after-free and double-free share common erroneous behaviors (i.e., at the moment when the unsafe dangling pointer is created).

SIGSEGV exceptions can be abused to launch control-flow hijacking attacks, information leaks, etc. They are particularly dangerous if the vulnerability offers a control between free and use (the right-most column, `Control b/w free and use`). For example, with this control, malicious JavaScript code can place crafted data in freed memory and turn the SIGSEGV exception (i.e., deference the unsafe dangling pointer) into control-flow hijacking attacks or information leakages depending on the context of dereference operations. Moreover, this control between free and use also implies that the attackers can bypass memory error detection tools (e.g., AddressSanitizer [101]) because it allows the attackers to force the reuse of a freed memory region (see more details in §2.7 and a concrete bypassing case (Figure 10)).

Once Chromium is instrumented with DANGNULL, all of these cases were safely mitigated (`Nullify-value` column). Depending on the nullify value provided as a parameter, all 28 cases (7 rows by 4 columns) result in the following three categories: 1) integer values represent that DANGNULL securely contained SIGSEGV exceptions with safe-dereference; 2) *stopped by assertion* represents that DANGNULL re-utilized existing safe assertions in Chromium; and 3) check marks (✓) represent that Chromium continuously runs as if Chromium is already patched.

For the safe-dereference cases, it is worth noting that the dereferenced address values are quite small (at most 0x2e8). Although these seven exploits would not be enough to represent all use-after-free behaviors, we believe this implies that the moderate size of null address padding for safe-dereference (§2.3.3.2) would be effective enough. DANGNULL's null address padding can be easily extended without actual physical memory overheads if necessary, and 64-bit x86 architectures can allow even more aggressive pre-mappings. Moreover, unlike the case before applying DANGNULL, these dereferenced addresses did not change between different executions. This indicates that unsafe dangling pointers were successfully nullified using DANGNULL and thus the random factor of the object's base address is canceled out.

In addition, the dereferenced address values show certain patterns depending on the nullify value. While address values are different when the nullify value is 0 (explained later in this section), address values generally show linear relationships with nullify values. For example, in Bug ID 261836, address values were {0x21, 0x22, 0x23} when nullify values were {1, 2, 3}, respectively. This pattern presents another clue that unsafe dangling pointers were actually nullified by DANGNULL because the difference of each address value and the nullify value is the same between different executions. We manually verified that the difference is actually the index offsets of the dereference operations by analyzing the crash call stacks and the source code. Bug ID 282088, which has 0xf0 for all different nullify values, was an exception. This is caused by aligned memory access in the dereference operation (i.e., the pointer is AND masked with 0xfffffff0).

As we mentioned in §2.3.3.2, DANGNULL can also utilize existing sanity checking code. These cases are shown as a check mark (✓) indicating that Chromium correctly handled the exploit as if it was patched. Because program developers usually insert null pointer checks before pointer dereferences, unsafe dangling pointers nullified by DANGNULL did not flow into the dereference instructions which would cause safe-dereference. Instead, it was handled as an expected error and Chromium displayed the same output as a later patched version. We admit that it would be premature to argue that unsafe dangling pointers nullified by DANGNULL can always be handled in this way, but we believe this is an interesting direction for future work. Existing research [78] also showed that skipping erroneous null-dereference instructions can keep the target application safely running. Besides normal checks, DANGNULL was also able to re-utilize existing null pointer assertions, as demonstrated in Bug ID 162835.

To clearly illustrate how DANGNULL mitigates these use-after-free exploits on the Chromium browser, Figure 7 shows the simplified vulnerable code snippet and its object relationships for Bug ID 286975. Two class objects hold mutual relationships with their own member variables. The root cause of the vulnerability is in that the in-bound pointer

Table 4: DANGNULL safely nullified all seven real-world use-after-free exploits for Chromium. Among these seven cases, three correctly run even after use-after-free exploits as if it was patched (represented as a ✓), and one is safely prevented as DANGNULL re-utilized existing assertions in Chromium (represented as *stopped by assertion*). Without DANGNULL, all exploits are potential threats, leading to control-flow hijacking attacks, information leakages, etc. To be concrete, we also include invalid pointers causing an exception with various nullification values (0-3), and their threat in terms of the chances of an attacker's control between free and use.

Bug ID	**CVE**	**Severity**	**No-Nullify**	**Nullify-value**				**Control b/w**
				0	**1**	**2**	**3**	**free and use**
261836	-	High	0x7f27000001a8	0x2e8	0x21	0x22	0x23	yes
265838	2013-2909	High	0x1bfc9901ece1	✓	0x1	0x2	0x3	yes
279277	2013-2909	High	0x7f2f57260968	✓	0x1	0x2	0x3	yes
282088	2013-2918	High	0x490341400000	0xf0	0xf0	0xf0	0xf0	difficult
286975	2013-2922	High	0x60b000006da4	✓	0x15	0x16	0x17	yes
295010	2013-6625	High	0x897ccce6951	0x30	0x1	0x2	0x3	yes
162835	2012-5137	High	0x612000046c18	*stopped by assertion*				yes

(`m_host`) of `HTMLTemplateElement` is not nullified when it is freed (patches are shown as + marked lines). As a result, use-after-free occurs when `containsIncludingHostElements()` dereferences `m_host`. Under DANGNULL, it immediately eliminates the root cause of the vulnerability by nullifying `m_host` when the object is freed. Thus, DANGNULL not only mitigated Use-After-Free, but also utilized the existing null pointer checks at line 24 and helped the browser run as if it were patched.

2.5.2 Instrumentation

To see how DANGNULL's static instrumentation changes the output binary, we measured the number of inserted instructions and the file sizes of both the original and instrumented binaries. The number of inserted instructions depends on the number of pointer propagation instructions as described in Figure 3. Accordingly, file size increments in instrumented binaries are proportional to the number of inserted instructions (excepting the 370KB runtime library).

The results for SPEC CPU 2006 benchmarks are shown in Table 5. A total of 16 programs were instrumented (eleven C programs and five C++ programs). The number of inserted instructions varied across each program. This variance is not only influenced by

Table 5: Details of instrumented binaries (the left half) and their runtime properties (the right half) in SPEC CPU2006. The left half describes the details of incremented file size due to newly inserted instrumentation instructions. The runtime library of DangNull is about 370 KB; DangNull requires approximately 40 B per instrumentation to trace pointer propagation. The right half represents the details of the programs' runtime behavior (e.g., increase of memory usage and the number of pointers and objects in each benchmark). The increase of memory (due to `shadowObjTree`) depends on the number of objects and pointers created and freed in total; `bzip2`, which has minimal memory allocation, imposed no extra memory overhead, while `gcc`, which has many memory operations, imposes about 80 MB of extra memory overhead with DangNull.

Name	File Size (KB)		# of instructions		# of objects		# of pointers		# Nullify	Memory (MB)	
	before	after	inserted	total	total	peak	total	peak		before	after
bzip2	172	549	13	15k	7	2	0	0	0	34	34
gcc	8k	9k	9k	606k	165k	3k	3167k	178k	104k	316	397
mcf	53	429	95	2k	2	1	0	0	0	569	570
milc	351	737	71	24k	38	33	0	0	0	2k	2k
namd	1k	1k	45	77k	964	953	0	0	0	44	114
gobmk	5k	6k	201	156k	12k	47	0	0	0	23	28
soplex	4k	4k	264	74k	1k	88	14k	172	140	7	14
povray	3k	3k	941	194k	15k	9k	7923k	26k	6k	38	81
hmmer	814	1k	94	60k	84k	28	0	0	0	1	18
sjeng	276	662	17	22k	1	1	0	0	0	171	171
libquantum	106	483	21	7k	49	2	0	0	0	0	2
h264ref	1k	1k	154	115k	9k	7k	906	111	101	44	208
lbm	37	411	9	2k	2	1	0	0	0	408	409
astar	195	574	54	8k	130k	5k	2k	148	20	13	135
sphinx3	541	931	170	34k	6k	703	814k	14k	0	46	62
xalancbmk	48k	51k	7k	645k	28k	4k	256k	18k	10k	7	76

the total number of instructions, but also by a program's characteristics (some programs have more pointer propagation than others). For example, although mcf has fewer total instructions, DangNull inserted more than 10 times the number of instructions into mcf than lbm, a similarly-sized application. The file size increments were proportional to the number of inserted instructions as expected after subtracting the size of the runtime library.

The result for the Chromium browser is shown in Table 6. A total of 140,000 instructions were inserted, which is less than 1% of the whole program. This implies that pointer propagation instructions appeared with less than 1% probability. The file size is increased by about 0.5%, for a total size of 1,868 MB. Note that the Chromium build process uses static linking, and thus the resultant binary includes all shared libraries. We believe the file size increment (0.5%) should not be a concern for distribution or management of the instrumented binary.

Table 6: Static instrumentation results on Chromium

Name	File Size (MB)			# of instructions	
	before	after	incr.	inserted	total
Chromium	1858	1868	10	140k	16,831k

2.5.3 Runtime Overheads

As DANGNULL must trace object relationships for nullification, it increases both execution time and memory usage. To determine how much runtime overhead DANGNULL imposes on target applications, we measured various runtime overheads of SPEC CPU2006 and the Chromium browser.

Figure 8 shows the runtime performance overheads of DANGNULL running SPEC CPU2006 benchmarks. The overheads largely depend on the number of objects and pointers that DANGNULL traced and stored in `shadowObjTree`. These metadata tracing measurements are shown in the right half of Table 5. As we described in §2.5.2, each application has a different number of object allocations and degree of pointer propagation. Accordingly, each object allocation and pointer propagation would insert extra metadata into `shadowObjTree` unless it fails runtime range analysis. DANGNULL imposed an average performance overhead of 80%. DANGNULL caused more runtime overhead if the application had to trace a large number of pointers. For example, in the povray case, a total of 7,923,000 pointers were traced because it maintains a large number of pointers to render image pixels, and thus increased execution time by 270% with 213% memory overhead. On the other hand, in h264ref, only 906 pointers were traced and resulted in a 1% increase in execution time and 472% memory overhead.

To obtain a practical measurement of performance impacts of DANGNULL, we also tested the DANGNULL-hardened Chromium browser to see how much DANGNULL would affect web navigation experiences. First, seven different browser benchmarks listed in Table 7 are tested. Among them, Octane 2.0 [53], SunSpider 1.0.2 [122], and Dromaeo JS [83] evaluate the JavaScript engine performance. Balls [117], Dromaeo DOM, JS Lib, and html5 [117] evaluate rendering performance. For the JavaScript engine benchmarks, DANGNULL showed only 4.8% averaged overhead. This is because heavy JavaScript computations are mostly executed in the form of JIT-compiled code, which is not instrumented by DANGNULL. For rendering benchmarks, DANGNULL showed 53.1% overhead on average, which implies that rendering computations (i.e., DOM or HTML related computations) were affected by DANGNULL to some extent.

Note that the actual overhead for most end-users would be the combined overheads of both JavaScript and rendering computations. Therefore, we also measured the page loading time for the Alexa top 100 websites [5], as this would be representative of the actual overhead an end user would experience. To measure the page loading time, we developed a simple Chromium extension which computes the time difference between the initial fetch event and the end of page loading event. Each website is visited three times while the browser is newly spawned and the user profile directory is deleted before each visit to avoid page cache effects. On average, DANGNULL showed 7% increased load time. DANGNULL showed 2326 ms page loading time with 317.6ms standard deviation. Original Chromium showed 2165 ms with 377.9ms standard deviation. Due to the variability of network traversal and the response time of web servers, most performance impacts of DANGNULL introduce no greater magnitude of load time variability than those introduced by factors unrelated to DANGNULL.

To illustrate how DANGNULL behaves for web page loading, Table 8 shows detailed overheads. Four popular websites are visited, of which two were logged in using active accounts. When visiting `youtube.com`, page load time increased 32.8% as it renders many images. For login pages, DANGNULL maintained a similar overhead ratio even though it traced a relatively large number of objects and pointers.

Table 7: Chromium Benchmark results with and without DangNull. High/low denotes whether performance was higher or lower when compared to the unmodified test. For JavaScript benchmarks, DangNull imposes negligible overheads, varying from 2.7-8.6%. For rendering benchmarks requiring lots of memory operations (e.g., allocating DOM elements), DangNull exhibits 22.2%-105.3% overhead depending on type.

Benchmarks	**JavaScript**			**Rendering**	
	Octane	**SunSpider**	**Dromaeo JS**	**Dromaeo DOM**	**Dromaeo JS Lib**
(unit, [high or low])	(score, high)	(ms, low)	(runs/s, high)	(runs/s, high)	(runs/s, high)
Original	13,874	320.0	1,602.1	857.8	216.0
DangNull	13,431	347.5	1,559.6	509.1	168.1
Slowdown	3.2%	8.6%	2.7%	40.7%	22.2%

Table 8: Page load time overhead when visiting four popular websites.

Website	**Action**	**Page Complexity**		**Loading Time (sec)**	
		# Req	# DOM	Original	DangNull
gmail.com	visit	13	164	0.49	0.60 (22.4%)
twitter.com	visit	14	628	1.05	1.16 (10.5%)
amazon.com	visit	264	1893	1.37	1.60 (16.8%)
youtube.com	visit	43	2293	0.61	0.81 (32.8%)
gmail.com	login	177	5040	6.40	7.66 (19.7%)
twitter.com	login	60	3124	2.16	2.77 (28.2%)

Table 9: Detailed runtime properties when visiting four popular websites.

Website	**Action**	**# of objects**		**# of pointers**		**# Nullify**	**Memory (MB)**	
		total	**peak**	**total**	**peak**		**before**	**after**
gmail.com	visit	123k	22k	32k	12k	7k	46	171
twitter.com	visit	121k	23k	35k	13k	8k	48	178
amazon.com	visit	166k	25k	81k	28k	16k	57	200
youtube.com	visit	127k	23k	46k	16k	9k	49	178
gmail.com	login	295k	31k	165k	49k	32k	96	301
twitter.com	login	172k	27k	71k	23k	15k	98	276

2.5.4 Compatibility

Due to the nullification of benign dangling pointers, it is possible that DANGNULL may cause false positives by changing program execution semantics (i.e., dangling pointers that will not be dereferenced, but their address values will be use). To see how much negative impact DANGNULL would impose on such program execution semantics, we used the Chromium browser as an example and ran stress tests on DANGNULL with various benign inputs. These tests include base unit tests, LayoutTests, Acid3 tests, and visiting Alexa top 500 websites. The first three tests are well known tests to verify the correctness of browser implementations, and we additionally visited Alexa top 500 websites to see if DANGNULL causes any issues when browsing popular websites.

Base unit tests [31], which are distributed with the Chromium source code, are regression tests that directly communicate with the native interfaces of Chromium and check whether basic functions are working correctly. DANGNULL passed all 1,100 test cases.

LayoutTests [31], which are also distributed with the Chromium source code, include 30,120 test cases. These tests are web page files (e.g., `.html`, `.js`, etc), and check whether the browser renders various web pages as expected. Test results for Chromium secured with DANGNULL were identical to original Chromium. Both passed 30,035 tests, but failed the same 85 tests. For the failed cases, four tests crashed the browser, 70 tests failed to generate expected output, and 11 tests caused a timeout. We manually verified the four crashing tests, and all were to-be-fixed bugs in the Chromium version we evaluated. As the set of failed tests were the same for original and instrumented Chromium, we believe DANGNULL would match the rendering conformity of original Chromium.

Acid3 tests [116] check whether the browser conforms with published web standards. It runs various JavaScript code and compares the rendered output with a reference input to see if there are any differences. DANGNULL passed the test with the full score (100/100).

We also visited Alexa top 500 websites [5] to see if DANGNULL introduces any unexpected behavior. Each time the browser visits a website, we waited 10 seconds and checked whether DANGNULL raised false alarms during the visit. For all top 500 websites, DANGNULL displayed all web pages without false alarms.

Based on this compatibility evaluation, we believe that DANGNULL would not impact original execution semantics in practice. We found one rare false positive case while manually handling the browser, which is described in §2.6.

2.6 Discussion

Usage scenarios. Because this thesis focuses on the correctness and effectiveness of DANGNULL against use-after-free and double-free vulnerabilities, we have not specifically restricted the possible usage scenarios of DANGNULL. In general, we believe DANGNULL can be applied to the following three scenarios:

- Back-end use-after-free detection: many zero-day attacks are based on use-after-free vulnerabilities [86]; DANGNULL can be used to detect these attacks. To the best of our knowledge, DANGNULL is the only mitigation system that can detect use-after-free exploits carefully developed for complex and large scale software (e.g., Chromium).
- Runtime use-after-free mitigation for end users: if performance overhead is not the primary concern of end users, DANGNULL is an effective use-after-free mitigation tool with moderate performance overhead, especially for web browsers.
- Use-after-free resilient programs: we have shown that DANGNULL can utilize existing sanity check routines and survive use-after-free attempts. By integrating automatic runtime repair work [78], we believe DANGNULL can evolve to support use-after-free resilient programs in the future.

Performance optimization. We believe DANGNULL's performance overhead can be further improved, especially for performance critical applications. First of all, instrumentation phases can be further optimized by leveraging more sophisticated static analysis. For example, if it is certain that the original code already nullifies a pointer, DANGNULL would not need to nullify it again. Although we have not heavily explored this direction, this has to be done carefully because soundly learning this information involves pointer-aliasing problems, which are well-known difficult problems in program analyses, and any incorrect analysis results would lead to both false positives and false negatives.

Moreover, we identified that manipulation of `shadowObjTree` is the main performance bottleneck, and this can be optimized by 1) leveraging transactional memory [56] to enhance locking performance on `shadowObjTree`; 2) designing a software cache for `shadowObjTree`; 3) using alternative data-structures to implement `shadowObjTree` (e.g., alignment-based metadata storage by replacing the memory allocator [54]); or 4) creating a dedicated analyzing thread or process for `shadowObjTree` [63].

False negatives. DANGNULL's static instrumentation assumes that a pointer is propagated only if either the left– or right–hand side of a variable is a pointer type. This would not be true if the program is written in a manner such that the propagation is done between non-pointer-typed variables. Consider the example we introduced in Figure 1. If the child member variable is typed as `long` (i.e., `long` child at line 4) and all the following operations regarding child are using type casting (i.e., doc->child=(long)body at line 13 and ((Elem*)doc->child)->getAlign() at line 21), then such a pointer propagation would not be traced. DANGNULL would under-trace object relationships in this case, and there would be false negatives if child becomes an unsafe dangling pointer.

False positives. To stop dereferencing on unsafe dangling pointers, DANGNULL nullifies not only unsafe dangling pointers but also benign dangling pointers. This implies DANGNULL additionally nullifies benign dangling pointers, and it is possible it may cause some false positives, although these should not have any semantic meaning as they are "dangled".

While testing DANGNULL for SPEC CPU benchmarks and the Chromium browser, we found one rare false positive case. This false positive case sporadically occurs when a new tab is manually created inside the Chromium browser, and it is related to the unique pointer hash table design (Figure 9).

We believe this false positive example would not be a critical concern for deploying DANGNULL due to its rareness. As we described in the compatibility evaluation in §2.5.4, DANGNULL passed more than 30,000 stress tests with the Chromium browser, a large scale and highly complicated application.

2.7 Related Work

Memory-related issues, including invalid memory accesses, memory leaks, and use-after-free bugs, have been studied for many years. Numerous methods have been proposed for C/C++ programs. In this section, we categorize them and compare them with DANGNULL.

Use-after-free detectors. There is a line of research specifically focusing on detecting use-after-free vulnerabilities. In general, use-after-free vulnerabilities can be detected through both static and dynamic analysis. However, since 1) a dangling pointer itself is not erroneous behavior and 2) statically determining whether a dangling pointer will actually be used in the future requires precise points-to and reachability analyses across all possible inter-procedure paths, even state-of-the-art use-after-free detection tools based on the static analysis are only suitable for analyzing small programs [45, 94].

For this reason, most use-after-free detectors [18, 87, 124] are based on the runtime dynamic analysis. CETS [87] maintains a unique identifier with each allocated object, associates this metadata with pointers, and checks that the object is still allocated on pointer dereferences. To handle pointer arithmetic, CETS uses taint propagation (i.e., the resulting pointer will inherit the metadata from the base pointer of the corresponding arithmetic operation). Unfortunately, the assumption behind this design choice —the propagated pointer should point to the same object—does not always hold, which results in high false positive rates. From our experiments, CETS raised false alarms on 5 out of 16 tested programs while DANGNULL was able to correctly run all 16 programs. In addition to high false positive rates, CETS relies on explicit checks to guarantee the memory access validity for all memory operations, thus imposing higher performance overhead in SPEC CPU2006 compared to DANGNULL. For 4 programs (bzip2, milc, sjeng, h264ref, and lbm) that CETS was able to run[7], on average it incurred 40% slow down, while DANGNULL incurred 1% slow down.

Undangle [18] is another runtime dynamic analysis tool to detect use-after-free. It assigns each return value of memory allocation functions a unique label, and employs a dynamic taint analysis to track the propagation of these labels. On memory deallocation, Undangle checks which memory slots are still associated with the corresponding label, and then determines the unsafe dangling pointers based on the *lifetime* of dangling pointers (i.e., if the lifetime of a dangling pointer is higher than the certain threshold number, it is identified as an unsafe dangling pointer). While this approach can collect more complete pointer propagation information than DANGNULL (which would better help a bug triage or debugging process), a significant performance cost is required.

Control flow integrity. Control flow integrity (CFI) [1, 125, 126, 129] enforces indirect function calls to be legitimate (i.e., enforcing integrity of the control-flows). Similarly, SafeDispatch [61] prevents illegal control flows from virtual function call sites. Unlike use-after-free and memory error detectors, CFI makes use-after-free vulnerabilities difficult to exploit. Specifically, CFI only shepherds function pointers to guarantee legitimate control flows. In practice, however, most CFI implementations enforce coarse-grained CFI to avoid heavy performance overheads and false positive alarms, but recent research [21, 40, 51, 52] has demonstrated that all the aforementioned coarse-grained CFI implementations can be bypassed. Moreover, dangling pointers can also be abused to corrupt non-control data (e.g., vector length variables, user privilege bits, or sandbox enforcing flags) in objects [25], all of which are not function pointers, which makes CFI based protection techniques bypassable. For example, a recent attack [80] overwrote user permission bits in the metadata to bypass user authorizations, including all other defense mechanisms. As an another example, vector

[7] CETS failed to compile 7 programs out of 16 SPEC CPU2006 programs we tested.

length variable corruption is one popular technique to exploit use-after-free vulnerabilities that lead to information leakage attacks or additional heap buffer overflows.

DANGNULL eliminates dangling pointers at the moment they are created. Thus, it can protect not only control flows but also any other security sensitive metadata in the objects from being abused by use-after-free or double-free vulnerabilities.

Memory error detectors. Memcheck (Valgrind) [91] and Purify [55] are popular solutions for detecting memory errors. Since their main goal is to help debugging, they are designed to be complete (incurring no false negatives) and general (detecting all classes of memory problems) in identifying memory-leak vulnerabilities, imposing very high memory and CPU overheads.

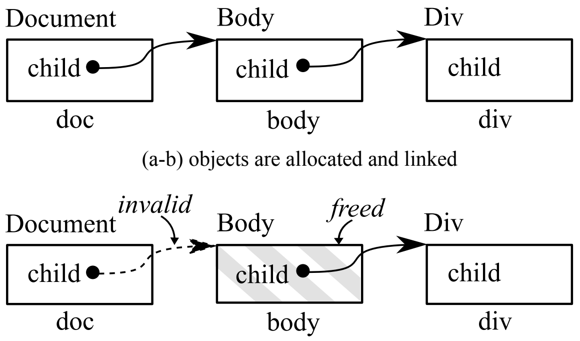
AddressSanitizer [101] is another popular tool developed recently that optimizes the method of representing and probing the status of allocated memory. However, due to an assumption to support this optimization (a quarantine zone that prevents reuse of previously freed memory blocks), it cannot detect use-after-free bugs if the assumption does not hold (i.e., heap objects are reallocated). Specifically, attackers can easily leverage various techniques to force reallocation of previously freed memory blocks, such as Heap Spraying [39, 97] and Heap Fengshui [105]. To clearly demonstrate this issue, we developed a proof-of-concept exploit bypassing the detection of AddressSanitizer (Figure 10). However, with DANGNULL, all dangling pointers will be nullified upon the deallocation of their objects, rendering use-after-free vulnerabilities unexploitable, even with sophisticated manipulations.

Safe memory allocators. Many safe memory allocators have been proposed to prevent dangling pointer issues. Cling [3] can disrupt a large class of exploits targeting use-after-free vulnerabilities by restricting memory reuse to objects of the same type. Diehard and Dieharder [12, 92] mitigate dangling pointer issues by approximating an infinite-sized heap.

Smart pointers. A smart pointer is an abstract data type that encapsulates a pointer to support automatic resource management. Theoretically, an application would not suffer from use-after-free issues if all the pointers are represented with smart pointers (i.e., no raw pointers are used in the application code). However, it is common to expose raw pointers even in applications heavily using smart pointers. For example, in order to break the resource graph cycle connected with shared pointers (e.g., `std::shared_ptr` in `C++11`), browser rendering engines including WebKit [117] and Blink [15] usually expose a raw pointer instead of using weak pointers (e.g., `std::weak_ptr` in `C++11`) to avoid extra performance overheads and be compatible with legacy code [79], and these exposed raw pointers have been major use-after-free vulnerability sources for those engines. Note that automatically wrapping raw pointers with smart pointers is another challenging static analysis problem, as this requires understanding precise raw pointer semantics to be properly implemented.

```
class Div: Element;
class Body: Element;
class Document {
  Element* child;
};

// (a) memory allocations
Document *doc = new Document();
Body *body = new Body();
Div *div = new Div();

// (b) using memory: propagating pointers
doc->child = body;
body->child = div;

// (c) memory free: doc->child is now dangled
delete body;

// (d) use-after-free: dereference the dangled pointer
if (doc->child)
    doc->child->getAlign();
```

(i) Vulnerable code



(a-b) objects are allocated and linked

(c-d) body is freed (so dangled), and doc reads the invliad memory

(ii) Objects and their relations in the course of running (i)

Figure 1: A running example of a use-after-free vulnerability. Document (`doc`), Body (`body`), and Div (`div`) are allocated and referencing each other. After `body` is freed by `delete`, `body` becomes a dangling pointer, pointing to an invalid memory region. Attempting to access `body` via `child` will lead to unsafe runtime behaviors.

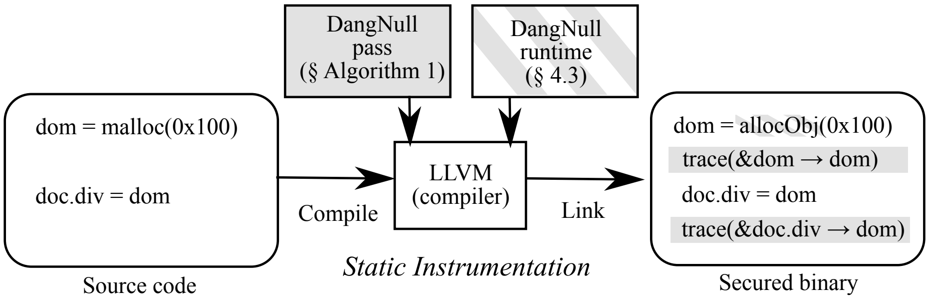


Figure 2: Overview of DANGNULL's design. DANGNULL consists of two main components, a static instrumentation (§2.3.2) and a runtime library (§2.3.3). At the static instrumentation stage, DANGNULL identifies propagation (their dependencies) of object pointers and inserts a call to the tracing routine to keep track of point-to relations between pointers and memory objects. At runtime, DANGNULL interposes all memory allocations to maintain the data structure for live objects (§2.3.3.1), and all memory frees to nullify all the pointers pointing to the object that is about to be freed (§2.3.3.2).

```
for function in Program:
   # All store instructions are
   # in the LLVM IR form of 'lhs := rhs'.
   for storeInstr in function.allStoreInstructions:
      lhs = storeInstr.lhs
      rhs = storeInstr.rhs

      # Only insterested in a pointer on the heap.
      if mustStackVar(lhs):
         continue
      if not isPointerType(rhs):
         continue

      new = createCallInstr(trace, lhs, rhs)
      storeInstr.appendInstr(new)
```

Figure 3: The algorithm for static instrumentation. For every store instruction where the destination may stay on the heap, DANGNULL inserts `trace()` to keep track of the relation between the pointer and the object it points to.

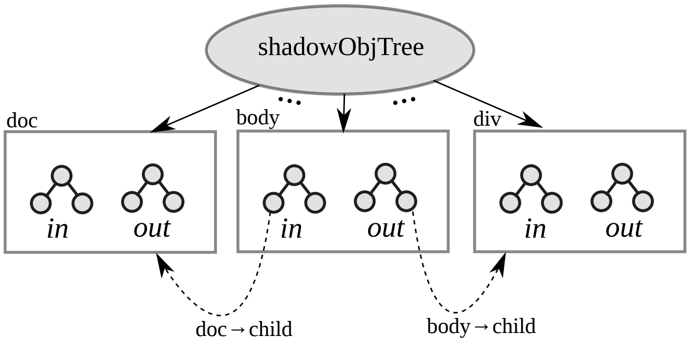


Figure 4: The shadow object tree and three shadow objects (`doc`, `body`, and `div`) corresponding to Figure 1. To simplify the representation, only in- and out- bound pointers of the `body` shadow object are shown. `body` keeps the in-bound pointer for `doc→child`, which points to the shadow object of `doc`, and the out-bound pointer for `body->child`, which points to the shadow object of `div`.

```
// (a) memory allocations
+ Document *doc = allocObj(Document);
+ Body *body = allocObj(Body);
+ Div *div = allocObj(Div);

// (b) using memory: propagating pointers
doc->child = body;
+ trace(&doc->child, body);

body->child = div;
+ trace(&body->child, div);

// (c) memory free: unsafe dangling pointer, doc->child,
//                 is automatically nullified
+ freeObj(body);

// (d) use-after-free is prevented, avoid dereferencing it
if (doc->child)
    doc->child->getAlign();
```

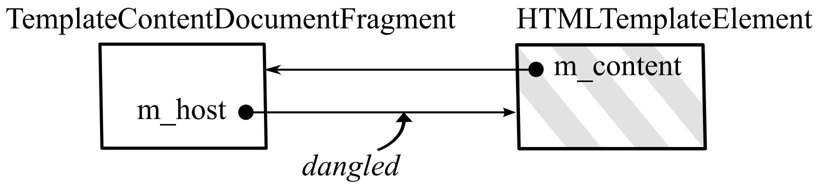
Figure 5: Instrumented running example of Figure 1 (actual instrumentation proceeds at the LLVM Bitcode level). Memory allocations (`new`) and deallocations (`free`) are replaced with `allocObj()` and `freeObj()`, and `trace()` is placed on every memory assignment, according to the static instrumentation algorithm (Figure 3).

```
def allocObj(size):
    ptr = real_alloc(size)
    shadowObj = createShadowObj(ptr, size)
    shadowObjTree.insert(shadowObj)
    return ptr

# NOTE. lhs <- rhs
def trace(lhs, rhs):
    lhsShadowObj = shadowObjTree.find(lhs)
    rhsShadowObj = shadowObjTree.find(rhs)

    # Check if lhs and rhs are eligible targets.
    if lhsShadowObj and rhsShadowObj:
        removeOldShadowPtr(lhs, rhs)
        ptr = createShadowPtr(lhs, rhs)
        lhsShadowObj.insertOutboundPtr(ptr)
        rhsShadowObj.insertInboundPtr(ptr)
    return

def freeObj(ptr):
    shadowObj = shadowObjTree.find(ptr)

    for ptr in shadowObj.getInboundPtrs():
        srcShadowObj = shadowObjTree.find(ptr)
        srcShadowObj.removeOutboundPtr(ptr)
        if shadowObj.base <= ptr < shadowObj.end:
            *ptr = NULLIFY_VALUE

    for ptr in shadowObj.getOutboundPtrs():
        dstShadowObj = shadowObjTree.find(ptr)
        dstShadowObj.removeInboundPtr(ptr)

    shadowObjTree.remove(shadowObj)

    return real_free(ptr)
```

Figure 6: The Runtime library algorithm. All error handling and synchronization code is omitted for clarity. DANGNULL has a global data structure (thread-safe), `shadowObjTree`, to maintain object relations with shadow objects. `allocObj()` and `freeObj()` replaced the `malloc()` and `free()` (and their equivalence, `new` and `delete` in `C++`), and `trace()` will be inserted on every memory assignments as a result of the static instrumentation (§2.3.2).

```
class TemplateContentDocumentFragment: public Node {
  const Element* m_host;
  // nullify its m_host
+ void clearHost() { m_host = NULL;}
};

class Element: public Node;

class HTMLTemplateElement: public Element {
  mutable RefPtr<Node> m_content;
  ~HTMLTemplateElement() {
    // nullify if m_content is alive
+   if (m_content)
+     m_content->clearHost();
  }
};

class Node {
  void containsIncludingHostElements(Node *node) {
    TemplateContentDocumentFragment *t = \
          (TemplateContentDocumentFragment*)node;

    // null-pointer check
    if (t->m_host)
      // dereference dangling pointer
      ((Node*)(t->m_host))->getFlags();
  }
};
```

(i) Simplified code snippet. Lines marked + are the vulnerability patch.



(ii) Objects and their relationships while running (i)

Figure 7: The vulnerable code and its object relationships for Bug ID 286975. `TemplateContentDocumentFragment` and `HTMLTemplateElement` have mutual references via `m_host` and `m_content`. In the vulnerable code, even after the `HTMLTemplateElement` object is freed, the `TemplateContentDocumentFragment` object still holds a reference to the object in `m_host`, resulting in a use-after-free vulnerability when `containsIncludingHostElements()` is invoked. With DANGNULL, use-after-free is mitigated, and Chromium continues to run correctly.

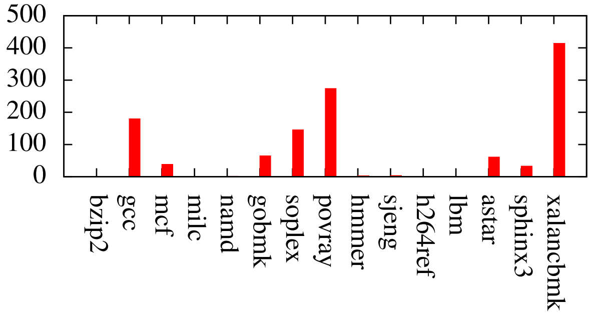


Figure 8: Runtime speed overheads when running SPEC CPU2006. y-axis shows slowdowns (%) of DangNull.

```
1  enum child_status {IN_USE=0, DELETED=1};
2  hash_map <Element*, child_status, ptrHash> allChilds;
3
4  Document *doc = new Document();
5  Element *elem = new Element();
6
7  // hold child reference
8  doc->child = elem;
9
10 // mark it as in-use in the hash_map
11 allChilds[elem] = IN_USE;
12
13 // delete child, nullified accordingly
14 delete doc->child;
15
16 // doc->child is nullified,
17 //   but Chromium relies on the stale pointer
18 allChilds[doc->child] = DELETED;
19
20 // makes sure all childs are deleted
21 for (it = allChilds.begin(); it != allChilds.end(); ++ it)
22   if (it->second == IN_USE)
23     delete it->first;
```

Figure 9: A simplified false positive example of DangNull in the Chromium browser. This sporadically occurred when the tab is manually created inside the browser. If applications rely on the stale pointer (using the freed pointer as a concrete value, as `doc→child` in line 18), DangNull can cause a false positive. We fixed this issue for DangNull by switching the order of deletion and marking operations (switching line 14 and line 18).

```
function onOpened() {
  buf = ms.addSourceBuffer(...);
  // disconnect the target obj
  vid.parentNode.removeChild(vid);
  vid = null;
  // free the target obj
  gc();

+  var drainBuffer = new Uint32Array(1024*1024*512);
+  drainBuffer = null;
+  // drain the quarantine zone
+  gc();

+  for (var i = 0; i < 500; i ++) {
+    // allocate/fill up the landing zone
+    var landBuffer = new Uint32Array(44);
+    for (var j = 0; j < 44; j ++)
+      landBuffer[j] = 0x1234;
+  }

  // trigger use-after-free
  buf.timestampOffset = 100000;
}

ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', onOpened);

// NOTE.
//   <video id="vid"></video>
vid = document.getElementById('vid');
vid.src = window.URL.createObjectURL(ms);
```

Figure 10: An example of exploits (Bug ID 162835) bypassing AddressSanitizer [101]. Lines with + marks show the bypassing routine, which keeps allocating the same sized memory to drain the quarantine zone of AddressSanitizer. Once the quarantine zone is drained, AddressSanitizer returns the previously freed memory block (i.e., an object is allocated in the previously freed memory region), which means it cannot identify memory semantic mismatches introduced by unsafe dangling pointers. Thus, AddressSanitizer cannot detect use-after-free exploits in this case, and this technique can be generalized to other use-after-free exploits with a different allocation size. However, DangNull detected this sophisticated exploit.

CHAPTER III

CAVER: VERIFYING TYPE CASTINGS TO DETECT BAD-CASTING

3.1 Eliminating Bad-Casting Vulnerability

The programming paradigm popularly known as object-oriented programming (OOP) is widely used for developing large and complex applications because it encapsulates the implementation details of data structures and algorithms into objects; this in turn facilitates cleaner software design, better code reuse, and easier software maintenance. Although there are many programming languages that support OOP, C++ has been the most popular, in particular when runtime performance is a key objective. For example, all major web browsers—Internet Explorer, Chrome, Firefox, and Safari are implemented in C++.

An important OOP feature is type casting that converts one object type to another. Type conversions play an important role in polymorphism. It allows a program to treat objects of one type as another so that the code can utilize certain general or specific features within the class hierarchy. Unlike other OOP languages—such as Java—that always verify the safety of a type conversion using runtime type information (RTTI), C++ offers two kinds of type conversions: `static_cast`, which verifies the correctness of conversion at *compile time*, and `dynamic_cast`, which verifies type safety at *runtime* using RTTI. `static_cast` is much more efficient because runtime type checking by `dynamic_cast` is an expensive operation (e.g., 90 times slower than `static_cast` on average). For this reason, many performance critical applications like web browsers, Chrome and Firefox in particular, prohibit `dynamic_cast` in their code and libraries, and strictly use `static_cast`.

However, the performance benefit of `static_cast` comes with a security risk because information at compile time is by no means sufficient to fully verify the safety of type conversions. In particular, upcasting (casting a derived class to its parent class) is always safe, but downcasting (casting a parent class to one of its derived classes) may not be safe because the derived class may not be a subobject of a truly allocated object in downcasting. Unsafe downcasting is better known as *bad-casting* or *type-confusion*.

Bad-casting has critical security implications. First, bad-casting is *undefined behavior* as specified in the C++ standard (5.2.9/11 [66]). Thus, compilers cannot guarantee the correctness of a program execution after bad-casting occurs (more detailed security implication analysis on undefined behavior is provided in §3.2). In addition to undefined behavior, bad-casting is similar to memory corruption vulnerabilities like stack/heap overflows and use-after-free. A bad-casted pointer violates a programmer's intended pointer semantics, and allows an attacker to corrupt memory beyond the true boundary of an object. For example, a bad-casting vulnerability in Chrome (CVE-2013-0912) was used to win the Pwn2Own 2013 competition by leaking and corrupting a security sensitive memory region [85]. More alarmingly, bad-casting is not only security-critical but is also common in applications. For example, 91 bad-casting vulnerabilities have been reported over the last four years in Chrome. Moreover, over 90% of these bad-casting bugs were rated as *security-high*, meaning that the bug can be directly exploited or indirectly used to mount arbitrary code execution attacks.

To avoid bad-casting issues, several C++ projects employ custom RTTI, which embeds code to manually keep type information at runtime and verify the type conversion safety of `static_cast`. However, only a few C++ programs are designed with custom RTTI, and supporting custom RTTI in existing programs requires heavy manual code modifications.

Another approach, as recently implemented by Google in the Undefined Behavior Sanitizer (UBSAN) [113], optimizes the performance of `dynamic_cast` and replaces all `static_cast` with `dynamic_cast`. However, this approach is limited because `dynamic_cast` only supports polymorphic classes, whereas `static_cast` is used for both polymorphic and non-polymorphic classes. Thus, this simple replacement approach changes the program semantics and results in runtime crashes when `dynamic_cast` is applied to non-polymorphic classes. It is difficult to identify whether a `static_cast` operation will be used for polymorphic or non-polymorphic classes without runtime information. For this reason, tools following this direction have to rely on manual *blacklists* (i.e., opt-out and do not check all non-polymorphic classes) to avoid runtime crashes. For example, UBSAN has to blacklist 250 classes, ten functions, and eight whole source files used for the Chromium browser [30], which is manually created by repeated trial-and-error processes. Considering the amount of code in popular C++ projects, creating such a blacklist would require massive manual engineering efforts.

This chapter presents CAVER, a runtime bad-casting detection tool that can be seamlessly integrated with large-scale applications such as commodity browsers. It takes a program's source code as input and automatically instruments the program to verify type castings at runtime. We designed a new metadata, the Type Hierarchy Table (`THTable`) to efficiently keep track of rich type information. Unlike RTTI, `THTable` uses a disjoint metadata scheme (i.e., the reference to an object's `THTable` is stored outside the object). This allows CAVER to overcome all limitations of previous bad-casting detection techniques: it not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. More specifically, CAVER achieves three goals:

- **Easy-to-deploy.** CAVER can be easily adopted to existing C++ programs without any manual effort. Unlike current state-of-the-art tools like UBSAN, it does not rely on manual blacklists, which are required to avoid program corruption. To demonstrate, we have integrated CAVER into two popular web browsers, Chromium and Firefox, by only modifying its build configurations.
- **Coverage.** CAVER can protect all type castings of both polymorphic and non-polymorphic classes. Compared to UBSAN, CAVER covers 241% and 199% more classes and their castings, respectively.

- **Performance.** CAVER also employs optimization techniques to further reduce runtime overheads (e.g., type-based casting analysis). Our evaluation shows that CAVER imposes up to 7.6% and 64.6% overheads for performance-intensive benchmarks on the Chromium and Firefox browsers, respectively. On the contrary, UBSAN is 13.8% slower than CAVER on the Chromium browser, and it cannot run the Firefox browser due to a runtime crash.

To summarize, this chapter makes the following contributions:

- **Security analysis of bad-casting.** We analyzed the bad-casting problem and its security implications in detail, thus providing security researchers and practitioners a better understanding of this emerging attack vector.
- **Bad-casting detection tool.** We designed and implemented CAVER, a general, automated, and easy-to-deploy tool that can be applied to any C++ application to detect (and mitigate) bad-casting vulnerabilities. We have shared CAVER with the Firefox team [1] and made our source code publicly available.
- **New vulnerabilities.** While evaluating CAVER, we discovered *eleven* previously unknown bad-casting vulnerabilities in two mature and widely-used open source projects, GNU `libstdc++` and Firefox. All vulnerabilities have been reported and fixed in these projects' latest releases. We expect that integration with unit tests and fuzzing infrastructure will allow CAVER to discover more bad-casting vulnerabilities in the future.

3.2 Bad-casting Demystified

Type castings in C++. Type casting in C++ allows an object of one type to be converted to another so that the program can use different features of the class hierarchy. C++ provides four explicit casting operations: `static`, `dynamic`, `const`, and `reinterpret`. In this thesis,

[1] The Firefox team at Mozilla asked us to share CAVER for regression testing on bad-casting vulnerabilities.

we focus on the first two types — static_cast and dynamic_cast (5.2.9 and 5.2.7 in ISO/IEC N3690 [66]) — because they can perform downcasting and result in bad-casting. static_cast and dynamic_cast have a variety of different usages and subtle issues, but for the purpose of this thesis, the following two distinctive properties are the most important: (1) time of verification: as the name of each casting operation implies, the correctness of a type conversion is checked (statically) at compile time for static_cast, and (dynamically) at runtime for dynamic_cast; (2) runtime support (RTTI): to verify type checking at runtime, dynamic_cast requires runtime support, called RTTI, that provides type information of the polymorphic objects.

Figure 11 illustrates typical usage of both casting operations and their correctness and safety: (1) casting from a derived class (pCanvas of SVGElement) to a parent class (pEle of Element) is valid *upcasting*; (2) casting from the parent class (pEle of Element) to the original allocated class (pCanvasAgain of SVGElement) is valid *downcasting*; (3) on the other hand, the casting from an object allocated as a base class (pDom of Element) to a derived class (p of SVGElement) is invalid *downcasting* (i.e., a bad-casting); (4) memory access via the invalid pointer (p->m_className) can cause memory corruption, and more critically, compilers cannot guarantee any correctness of program execution after this incorrect conversion, resulting in *undefined behavior*; and (5) by using dynamic_cast, programmers can check the correctness of type casting at runtime, that is, since an object allocated as a base class (pDom of Element) cannot be converted to its derived class (SVGElement), dynamic_cast will return a NULL pointer and the error-checking code (line 18) can catch this bug, thus avoiding memory corruption.

Type castings in practice. Although dynamic_cast can guarantee the correctness of type casting, it is an expensive operation because parsing RTTI involves recursive data structure traversal and linear string comparison. From our preliminary evaluation, dynamic_cast is, on average, 90 times slower than static_cast on average. For large applications such as the Chrome browser, such performance overhead is not acceptable: simply launching Chrome

```
1  class SVGElement: public Element { ... };
2
3  Element *pDom = new Element();
4  SVGElement *pCanvas = new SVGElement();
5
6  // (1) valid upcast from pCanvas to pEle
7  Element *pEle = static_cast<Element*>(pCanvas);
8  // (2) valid downcast from pEle to pCanvasAgain (== pCanvas)
9  SVGElement *pCanvasAgain = static_cast<SVGElement*>(pEle);
10
11 // (3) invalid downcast (-> undefined behavior)
12 SVGElement *p = static_cast<SVGElement*>(pDom);
13 // (4) leads to memory corruption
14 p->m_className = "my-canvas";
15
16 // (5) invalid downcast with dynamic_cast, but no corruption
17 SVGElement *p = dynamic_cast<SVGElement*>(pDom);
18 if (p) {
19     p->m_className = "my-canvas";
20  }
```

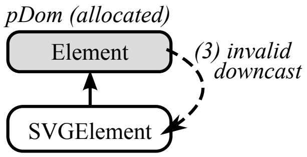


Figure 11: Code example using `static_cast` to convert types of object pointers (e.g., `Element` ↔ `SVGElement` classes). (1) is valid *upcast* and (2) is valid *downcast*. (3) is an invalid *downcast*. (4) Memory access via the invalid pointer result in memory corruption; more critically, compilers cannot guarantee the correctness of program execution after this incorrect conversion, resulting in *undefined behavior*. (5) Using `dynamic_cast`, on the other hand, the program can check the correctness of *downcast* by checking the returned pointer.

incurs over 150,000 casts. Therefore, despite its security benefit, the use of `dynamic_cast` is strictly forbidden in Chrome development.

A typical workaround is to implement custom RTTI support. For example, most classes in WebKit-based browsers have an `isType()` method (e.g., `isSVGElement()`), which indicates the true allocated type of an object. Having this support, programmers can decouple a `dynamic_cast` into an explicit type check, followed by `static_cast`. For example, to prevent the bad-casting (line 12) in Figure 11, the program could invoke the `isSVGElement()` method to check the validity of casting. However, this sort of type tracking and verification has to be manually implemented, and thus supporting custom RTTI in existing complex programs is a challenging problem. Moreover, due to the error-prone nature of manual modifications (e.g., incorrectly marking the object identity flag, forgetting to check the identity using `isType()` function, etc.), bad-casting bugs still occur despite custom RTTI [111].

Security implications of bad-casting. The C++ standard (5.2.9/11 [66]) clearly specifies that the behavior of an application becomes *undefined* after an incorrect static_cast. Because undefined behavior is an enigmatic issue, understanding the security implications and exploitability of bad-casting requires deep knowledge of common compiler implementations.

Generally, bad-casting vulnerabilities can be exploited via several means. An incorrectly casted pointer will either have wider code-wise visibility (e.g., allowing out-of-bound memory accesses), or become incorrectly adjusted (e.g., corrupting memory semantics because of misalignment). For example, when bad-casting occurs in proximity to a virtual function table pointer (vptr), an attacker can directly control input to the member variable (e.g., by employing abusive memory allocation techniques such as heap-spray techniques [39, 105]), overwrite the vptr and hijack the control flow. Similarly, an attacker can also exploit bad-casting vulnerabilities to launch non-control-data attacks [25].

The exploitability of a bad-casting bug depends on whether it allows attackers to perform out-of-bound memory access or manipulate memory semantics. This in turn relies on the details of object data layout as specified by the C++ application binary interface (ABI). Because the C++ ABI varies depending on the platform (e.g., Itanium C++ ABI [33] for Linux-based platforms and Microsoft C++ ABI [32] for Windows platforms), security implications for the same bad-casting bug can be different. For example, bad-casting may not crash, corrupt, or alter the behavior of an application built against the Itanium C++ ABI because the base pointer of both the base class and derived class always point to the same location of the object under this ABI. However, the same bad-casting bug can have severe security implications for other ABI implementations that locate a base pointer of a derived class differently from that of a base class, such as HP and legacy g++ C++ ABI [34]. In short, given the number of different compilers and the various architectures supported today, we want to highlight that bad-casting should be considered as a serious security issue. This argument is also validated from recent correspondence with the Firefox security team: after we reported two new bad-casting vulnerabilities in Firefox [10], they also pointed out the C++ ABI compatibility issue and rated the vulnerability as security-high.

Running example: CVE-2013-0912. Our illustrative Figure 11 is extracted from a real-world bad-casting vulnerability—CVE-2013-0912, which was used to exploit the Chrome web browser in the Pwn2Own 2013 competition. However, the complete vulnerability is more complicated as it involves a multitude of castings (between siblings and parents).

In HTML5, an SVG image can be embedded directly into an HTML page using the `<svg>` tag. This tag is implemented using the `SVGElement` class, which inherits from the `Element` class. At the same time, if a web page happens to contain unknown tags (any tags other than standard), an object of the `HTMLUnknownElement` class will be created to represent this unknown tag. Since both tags are valid HTML elements, objects of these types can be safely casted to the `Element` class. Bad-casting occurs when the browser needs to render an SVG image. Given an `Element` object, it tries to downcast the object to `SVGElement` so the caller function can invoke member functions of the `SVGElement` class. Unfortunately, since not all `Element` objects are initially allocated as `SVGElement` objects, this `static_cast` is not always valid. In the exploit demonstrated in the Pwn2Own 2013 competition [85], attackers used an object allocated as `HTMLUnknownElement`. As the size of an `SVGElement` object (160 bytes) is much larger than an `HTMLUnknownElement` object (96 bytes), this incorrectly casted object pointer allowed the attackers to access memory beyond the real boundary of the allocated `HTMLUnknownElement` object. They then used this capability to corrupt the `vtable` pointer of the object adjacent to the `HTMLUnknownElement` object, ultimately leading to a control-flow hijack of the Chrome browser. This example also demonstrates why identifying bad-casting vulnerabilities is not trivial for real-world applications. As shown in Figure 12, the `HTMLUnknownElement` class has more than 56 siblings and the `Element` class has more than 10 parent classes in WebKit. Furthermore, allocation and casting locations are far apart within the source code. Such complicated class hierarchies and disconnections between allocation and casting sites make it difficult for developers and static analysis techniques to reason about the true allocation types (i.e., alias analysis).

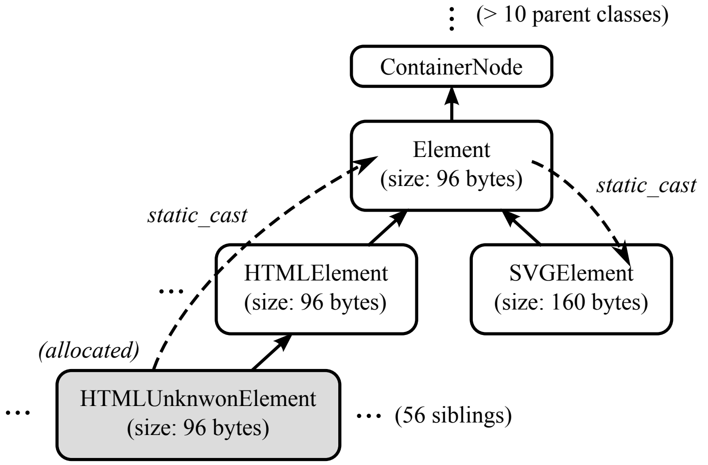


Figure 12: Inheritance hierarchy of classes involved in the CVE-2013-0912 vulnerability. MWR Labs exploited this vulnerability to hijack the Chrome browser in the Pwn2Own 2013 competition [85]. The object is allocated as `HTMLUnknownElement` and eventually converted (`static_cast`) to `SVGElement`. After this incorrect type casting, accessing member variables via this object pointer will cause memory corruption.

3.3 CAVER *Overview*

In this thesis, we focus on the correctness and effectiveness of CAVER against bad-casting bugs, and our main application scenario is as a back-end testing tool for detecting bad-casting bugs. CAVER's workflow (Figure 13) is as simple as compiling a program with one extra compile and link flag (i.e., `-fcaver` for both). The produced binary becomes capable of verifying the correctness of every type conversion at runtime. When CAVER detects an incorrect type cast, it provides detailed information of the bad-cast: the source class, the destination class, the truly allocated class, and call stacks at the time the bad-cast is captured. Figure 14 shows a snippet of the actual report of CVE-2013-0912. Our bug report experience showed that the report generated by CAVER helped upstream maintainers easily understand, confirm, and fix eleven newly discovered vulnerabilities without further examination.

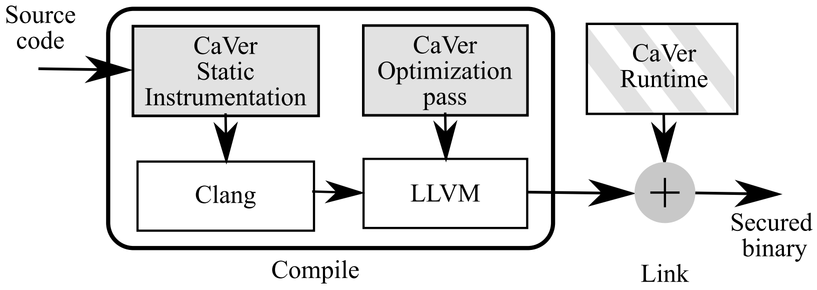


Figure 13: Overview of CAVER's design and workflow. Given the source code of a program, CAVER instruments possible castings at compile time, and injects CAVER's runtime to verify castings when they are performed.

```
== CaVer : (Stopped) A bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of 'blink::HTMLUnknownElement'
    from 'blink::Element
    to   'blink::SVGElement'
         Pointer         0x60c000008280
         Alloc base      0x60c000008280
         Offset          0x000000000000
         THTable         0x7f7963aa20d0

#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

Figure 14: A report that CAVER generated on CVE-2013-0912.

3.4 Design

In this section, we introduce the design of CAVER. We first describe how the THTable is designed to generally represent the type information for both polymorphic and non-polymorphic classes (§3.4.1), and then explain how CAVER associates the THTable with runtime objects (§3.4.2). Next, we describe how CAVER verifies the correctness of type castings (§3.4.3). At the end of this section, we present optimization techniques used to reduce the runtime overhead of CAVER (§3.4.4).

3.4.1 Type Hierarchy Table

To keep track of the type information required for validating type casting, CAVER incorporates a new metadata structure, called the *Type Hierarchy Table* (THTable). Given a pointer to an object allocated as type T, the THTable contains the set of all possible types to which T can be casted. In C++, these possible types are a product of two kinds of class relationships: *is-a* and *has-a*. The *is-a* relationship between two objects is implemented as class inheritance, the *has-a* relationship is implemented as class composition (i.e., having a member variable in a class). Thus, for each class in a C++ program, CAVER creates a corresponding THTable that includes information about both relationships.

To represent class inheritance, the THTable employs two unique design decisions. First, information on inherited classes (i.e., base classes) is unrolled and serialized. This allows CAVER to efficiently scan through a set of base classes at runtime while standard RTTI requires recursive traversal. Second, unlike RTTI, which stores a mangled class name, the THTable stores the hash value of a class name. This allows CAVER to avoid expensive string equality comparisons. Note, since all class names are available to CAVER at compile time, all possible hash collisions can be detected and resolved to avoid false negatives during runtime. Moreover, because casting is only allowed between classes within the same inheritance chain, we only need to guarantee the uniqueness of hash values within a set of those classes, as opposed to guaranteeing global uniqueness.

The THTable also includes information of whether a base class is a phantom class, which cannot be represented based on RTTI and causes many false alarms in RTTI-based type verification solutions [30]. We say a class P is a phantom class of a class Q if two conditions are met: (1) Q is directly or indirectly derived from P; and (2) compared to P, Q does not have additional member variables or different virtual functions. In other words, they have the same data layout. Strictly speaking, allocating an object as P and downcasting it to Q is considered bad-casting as Q is not a base class of P. However, such bad-castings are harmless from a security standpoint, as the pointer semantic after downcasting is the same. More importantly, phantom classes are often used in practice to implement object relationships with empty inheritances. For these reasons, CAVER deliberately allows bad-castings caused by phantom classes. This is done by reserving a one bit space in the THTable for each base class, and marking if the base class is a phantom class. We will describe more details on how the phantom class information is actually leveraged in §3.4.3.

In addition, the THTable contains information on composited class(es) to generally represent the type information for both polymorphic and non-polymorphic classes and overcome the limitation of RTTI-based type verification solutions. RTTI-based solutions locate a RTTI reference via the virtual function table (VTable). However, since only polymorphic classes have VTable, these solutions can cause runtime crashes when they try to locate the VTable for non-polymorphic classes. Unlike RTTI, CAVER binds THTable references to the allocated object with external metadata (refer §3.4.2 for details). Therefore, CAVER not only supports non-polymorphic objects, but it also does not break the C++ ABI. However, composited class(es) now share the same THTable with their container class. Since a composited class can also have its own inheritances and compositions, we do not unroll information about composited class(es); instead, CAVER provides a reference to the composited class's THTable. The THTable also stores the layout information (offset and size) of each composited class to determine whether the given pointer points to a certain composited class.

Other than inheritance and composition information as described above, the THTable contains basic information on the corresponding type itself: a type size to represent object ranges; and a type name to generate user-friendly bad-casting reports.

3.4.2 Object Type Binding

To verify the correctness of type casting, CAVER needs to know the actual allocated type of the object to be casted. In CAVER, we encoded this type information in the THTable. In this subsection, we describe how CAVER binds the THTable to each allocated object. To overcome the limitations of RTTI-based solutions, CAVER uses a disjoint metadata scheme (i.e., the reference to an object's THTable is stored outside the object). With this unique metadata management scheme, CAVER not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. Overall, type binding is done in two steps. First, CAVER instruments each allocation site of an application to pass the allocation metadata to its runtime library. Second, CAVER's runtime library maintains the allocation metadata and supports efficient lookup operations.

Instrumentation The goal of the instrumentation is to pass all information of an allocated object to the runtime library. To bind a `THTable` to an object, the runtime library needs two pieces of information: a reference to the `THTable` and the base address of the allocated object.

In C++, objects can be allocated in three ways: in heap, on stack, or as global objects. In all three cases, the type information of the allocated object can be determined statically at compile time. This is possible because C++ requires programmers to specify the object's type at its allocation site, so the corresponding constructor can be invoked to initialize memory. For global and stack objects, types are specified before variable names; and for heap objects, types are specified after the `new` operator. Therefore, CAVER can obtain type information by statically inspecting the allocation site at compile time. Specifically, CAVER generates the `THTable` (or reuses the corresponding `THTable` if already generated) and passes the reference of the `THTable` to the runtime library. An example on how CAVER instruments a program is shown in Figure 15.

For heap objects, CAVER inserts one extra function invocation (`trace_heap()` in Figure 15) to the runtime library after each `new` operator, and passes the information of the object allocated by `new`; a reference to the `THTable` and the base address of an object. A special case for the `new` operator is an array allocation, where a set of objects of the same type are allocated. To handle this case, we add an extra parameter to inform the runtime library on how many objects are allocated together at the base address.

Unlike heap objects, stack objects are implicitly allocated and freed. To soundly trace them, CAVER inserts two function calls for each stack object at the function prologue and epilogue (`trace_stack_begin()` and `trace_stack_end()` in Figure 15), and passes the same information of the object as is done for heap objects. A particular challenge is that, besides function returns, a stack unwinding can also happen due to exceptions and

```
  // Heap objects (dynamically allocated)
  void func_heap_ex() {
    C *p_heap_var = new C;
    C *p_heap_array = new C[num_heap_array];
+   trace_heap(&THTable(C), p_heap_var, 1);
+   trace_heap(&THTable(C), p_heap_array, num_heap_array);
    ...
  }

  // Stack objects
  void func_stack_ex() {
    C stack_var;
+   trace_stack_begin(&THTable(C), &stack_var, 1);
    ...
+   trace_stack_end(&stack_var);
  }

  // Global objects
  C global_var;

  // @.ctors: (invoked at the program's initialization)
  //   trace_global_helper_1() and trace_global_helper_2()
+ void trace_global_helper_1() {
+   trace_global(&THTable(C), &global_var, 1);
+ }

   // Verifying the correctness of a static casting
  void func_verify_ex() {
    B *afterAddr = static_cast<A>(beforeAddr);
+   verify_cast(beforeAddr, afterAddr, type_hash(A));
  }
```

Figure 15: An example of how CAVER instruments a program. Lines marked with + represent code introduced by CAVER, and &THTable(T) denotes the reference to the THTable of class T. In this example, we assume that the THTable of each allocated class has already been generated by CAVER.

setjmp/longjmp. To handle these cases, CAVER leverages existing compiler functionality (e.g., EHScopeStack::Cleanup in clang) to guarantee that the runtime library is always invoked once the execution context leaves the given function scope.

To pass information of global objects to the runtime library, we leverage existing program initialization procedures. In ELF file format files [119], there is a special section called .ctors, which holds constructors that must be invoked during an early initialization of a program. Thus, for each global object, CAVER creates a helper function (trace_global_helper_1() in Figure 15) that invokes the runtime library with static metadata (the reference to the THTable) and dynamic metadata (the base address and the number of array elements). Then, CAVER adds the pointer to this helper function to the .ctors section so that the metadata can be conveyed to the runtime library[2].

[2] Although the design detail involving .ctors section is platform dependent, the idea of registering the

Runtime library The runtime library of CAVER maintains all the metadata (THTable and base address of an object) passed from tracing functions during the course of an application execution. Overall, we consider two primary requirements when organizing the metadata. First, the data structure must support range queries (i.e., given a pointer pointing to an address within an object ([base, base+size)) CAVER should be able to find the corresponding THTable of the object). This is necessary because the object pointer does not always point to the allocation base. For example, the pointer to be casted can point to a composited object. In case of multi-inheritance, the pointer can also point to one of the parent classes. Second, the data structure must support efficient store and retrieve operations. CAVER needs to store the metadata for every allocation and retrieve the metadata for each casting verification. As the number of object allocations and type conversions can be huge (see §3.6), these operations can easily become the performance bottleneck.

We tackle these challenges using a hybrid solution We use red-black trees to trace global and stack objects and an alignment-based direct mapping scheme to trace heap objects[3].

We chose red-black trees for stack and global objects for two reasons. First, tree-like data structures are well known for supporting efficient range queries. Unlike hash-table-based data structures, tree-based data structures arrange nodes according to the order of their keys, whose values can be numerical ranges. Since nodes are already sorted, a balanced tree structure can guarantee $O(logN)$ complexity for range queries while hash-table-based data structure requires $O(N)$ complexity. Second, we specifically chose red-black trees because there are significantly more search operations than update operations (i.e., more type conversion operations than allocations, see §3.6), thus red-black trees can excel in performance due to self-balancing.

In CAVER, each node of a red-black tree holds the following metadata: the base address

helper function into the initialization function list can be generalized for other platforms as others also support .ctors-like features

[3] The alignment-based direct mapping scheme can be applied for global and stack objects as well, but this is not implemented in the current version. More details can be found in §3.7.

and the allocation size as the key of the node, and the THTable reference as the value of the node.

For global object allocations, metadata is inserted into the global red-black tree when the object is allocated at runtime, with the key as the base address and the allocation size[4], and the value as the address of the THTable. We maintain a per-process global red-black tree without locking mechanisms because there are no data races on the global red-black tree in CAVER. All updates on the global red-black tree occur during early process start-up (i.e., before executing any user-written code) and update orders are well serialized as listed in the .ctors section.

For stack object allocations, metadata is inserted to the stack red-black tree similar to the global object case. Unlike a global object, we maintain a *per-thread* red-black tree for stack objects to avoid data races in multi-threaded applications. Because a stack region (and all operations onto this region) are exclusive to the corresponding thread's execution context, this per-thread data structure is sufficient to avoid data races without locks.

For heap objects, we found that red-black trees are not a good design choice, especially for multi-threaded programs. Different threads in the target programs can update the tree simultaneously, and using locks to avoid data races resulted in high performance overhead, as data contention occured too frequently. Per-thread red-black trees used for stack objects are not appropriate either, because heap objects can be shared by multiple threads. Therefore, we chose to use a custom memory allocator that can support alignment-based direct mapping schemes [4, 54]. In this scheme, the metadata can be maintained for a particular object, and can be retrieved with $O(1)$ complexity on the pointer pointing to anywhere within the object's range.

[4] The allocation size is computed by multiplying the type size represented in THTable and the number of array elements passed during runtime.

3.4.3 Casting Safety Verification

This subsection describes how CAVER uses traced information to verify the safety of type casting. We first describe how the instrumentation is done at compile time, and then describe how the runtime library eventually verifies castings during runtime.

Instrumentation CAVER instruments `static_cast` to invoke a runtime library function, `verify_cast()`, to verify the casting. Here, CAVER analyzes a type hierarchy involving source and destination types in `static_cast` and only instruments for downcast cases. When invoking `verify_cast()`, CAVER passes the following three pieces of information: `beforeAddr`, the pointer address before the casting; `afterAddr`, the pointer address after the casting; and `TargetTypeHash`, the hash value of the destination class to be casted to (denoted as `type_hash(A)` in Figure 15).

Runtime library The casting verification is done in two steps: (1) locating the corresponding `THTable` associated with the object pointed to by `beforeAddr`; and (2) verifying the casting operation by checking whether `TargetTypeHash` is a valid type where `afterAddr` points.

To locate the corresponding `THTable`, we first check the data storage membership because we do not know how the object `beforeAddr` points to is allocated. Checks are ordered by their expense, and the order is critical for good performance. First, a stack object membership is checked by determinig whether the `beforeAddr` is in the range between the stack top and bottom; then, a heap object membership is checked by whether the `beforeAddr` is in the range of pre-mapped address spaces reserved for the custom allocator; finally a global object membership is checked with a bit vector array for each loaded binary module. After identifying the data storage membership, CAVER retrieves the metadata containing the allocation `base` and the reference to the `THTable`. For stack and global objects, the corresponding red-black tree is searched. For heap objects, the metadata is retrieved from the custom heap.

Next, CAVER verifies the casting operation. Because the `THTable` includes all possible types that the given object can be casted to (i.e., all types from both inheritances and compositions), CAVER exhaustively matches whether `TargetTypeHash` is a valid type where `afterAddr` points. To be more precise, the `afterAddr` value is adjusted for each matching type. Moreover, to avoid false positives due to a phantom class, CAVER tries to match all phantom classes of the class to be casted to.

3.4.4 Optimization

Since performance overhead is an important factor for adoption, CAVER applies several optimization techniques. These techniques are applied in two stages, as shown in Figure 13. First, offline optimizations are applied to remove redundant instrumentations. After that, additional runtime optimizations are applied to further reduce the performance overhead.

Safe-allocations Clearly, not all allocated objects will be involved in type casting. This implicates that CAVER does not need to trace type information for objects that would never be casted. In general, soundly and accurately determining whether objects allocated at a given allocation site will be casted is a challenging problem because it requires sophisticated static points-to analysis. Instead, CAVER takes a simple, yet effective, optimization approach inspired from `C` type safety checks in `CCured` [89]. The key idea is that the following two properties always hold for downcasting operations: (1) bad-casting may happen only if an object is allocated as a child of the source type or the source type itself; and (2) bad-casting never happens if an object is allocated as the destination type itself or a child of the destination type. This is because `static_cast` guarantees that the corresponding object must be a derived type of the source type. Since CAVER can observe all allocation sites and downcasting operations during compilation, it can recursively apply the above properties to identify *safe-allocation* sites, i.e., the allocated objects will never cause bad-casting.

Caching verification results Because casting verification involves loops (over the number of compositions and the number of bases) and recursive checks (in a composition case), it can be a performance bottleneck. A key observation here is that the verification result is always the same for the same allocation type and the same target type (i.e., when the type of object pointed by `afterAddr` and `TargetTypeHash` are the same). Thus, in order to alleviate this potential bottleneck, we maintain a cache for verification results, which is inspired by UBSAN [113]. First, a verification result is represented as a concatenation of the address of a corresponding `THTable`, the offset of the `afterAddr` within the object, and the hash value of target type to be casted into (i.e., &`THTable` || offset || TargetTypeHash). Next, this concatenated value is checked for existence in the cache before `verify_cast()` actually performs verification. If it does, `verify_cast()` can conclude that this casting is correct. Otherwise, `verify_cast()` performs actual verification using the `THTable`, and updates the cache only if the casting is verified to be correct.

3.5 Implementation

We implemented CAVER based on the LLVM Compiler project [114] (revision 212782, version 3.5.0). The static instrumentation module is implemented in Clang's `CodeGen` module and LLVM's `Instrumentation` module. The runtime library is implemented using the `compiler-rt` module based on LLVM's Sanitizer code base. In total, CAVER is implemented in 3,540 lines of C++ code (excluding empty lines and comments).

CAVER is currently implemented for the Linux x86 platform, and there are a few platform-dependent mechanisms. For example, the type and tracing functions for global objects are placed in the `.ctors` section of ELF. As these platform-dependent features can also be found in other platforms, we believe CAVER can be ported to other platforms as well. CAVER interposes threading functions to maintain thread contexts and hold a per-thread red-black tree for stack objects. CAVER also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack. We also modified the front-end drivers of Clang so that users of CAVER can easily build and secure their target applications with one extra compilation flag and linker flag, respectively.

3.6 Evaluation

We evaluated CAVER with two popular web browsers, Chromium [110] (revision 295873) and Firefox [115] (revision 213433), and two benchmarks from SPEC CPU2006 [107][5]. Our evaluation aims to answer the following questions:

- How easy is it to deploy CAVER to applications? (§3.6.1)
- What are the new vulnerabilities CAVER found? (§3.6.2)
- How precise is CAVER's approach in detecting bad-casting vulnerabilities? (§3.6.3)
- How good is CAVER's protection coverage? (§3.6.4)
- What are the instrumentation overheads that CAVER imposes and how many type castings are verified by CAVER? (§3.6.5)
- What are the runtime performance overheads that CAVER imposes? (§3.6.6)

Comparison methods We used UBSAN, the state-of-art tool for detecting bad-casting bugs, as our comparison target of CAVER. Also, We used CAVER-NAIVE, which disabled the two optimization techniques described in §3.4.4, to show their effectiveness on runtime performance optimization.

Experimental setup All experiments were run on Ubuntu 13.10 (Linux Kernel 3.11) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

3.6.1 Deployments

As the main design goal for CAVER is automatic deployments, we describe our experience of applying CAVER to tested programs including SPEC CPU 2006 benchmarks, the Chromium browser, and the Firefox browser. CAVER was able to successfully build and run these programs without any program-specific understanding of the code base. In particular, we added one line to the build configuration file to build SPEC CPU 2006, 21 lines to the `.gyp` build configuration to build the Chromium browser, and 10 lines to the `.mozconfig` build

[5] Although CAVER was able to correctly run all C++ benchmarks in SPEC CPU2006, only 483.xalancbmk and 450.soplex have downcast operations.

configuration file to build the Firefox browser. Most of these build configuration changes were related to replacing `gcc` with `clang`.

On the contrary, UBSAN crashed while running xalancbmk in SPEC CPU 2006 and while running the Firefox browser due to checks on non-polymorphic classes. UBSAN also crashed the Chromium browser without blacklists, but was able to run once we applied the blacklists provided by the Chromium project [30]. In particular, to run Chromium, the blacklist has 32 different rules that account for 250 classes, ten functions, and eight whole source files. Moreover, this blacklist has to be maintained constantly as newly introduced code causes new crashes in UBSAN [29]. This is a practical obstacle for adopting UBSAN in other C++ projects—although UBSAN has been open sourced for some time, Chromium remains the only major project that uses UBSAN, because there is a dedicated team to maintain its blacklist.

3.6.2 Newly Discovered Bad-casting Vulnerabilities

To evaluate CAVER's capability of detecting bad-casting bugs, we ran CAVER-hardened Chromium and Firefox with their regression tests (mostly checking functional correctness). During this evaluation, CAVER found eleven previously unknown bad-casting vulnerabilities in GNU `libstdc++` while evaluating Chromium and Firefox. Table 10 summarizes these vulnerabilities including related class information: allocated type, source, and destination types in each bad-casting. In addition, we further analyzed their security impacts: potential compatibility problems due to the C++ ABI (see §3.2) or direct memory corruption, along with security ratings provided by Mozilla for Firefox.

CAVER found two vulnerabilities in the Firefox browser. The Firefox team at Mozilla confirmed and fixed these, and rated both as *security-high*, meaning that the vulnerability can be abused to trigger memory corruption issues. These two bugs were casting the pointer into a class which is not a base class of the originally allocated type. More alarmingly, there were type semantic mismatches after the bad-castings—subsequent code could dereference

Table 10: A list of vulnerabilities newly discovered by CAVER. All security vulnerabilities listed here are confirmed, and already fixed by the corresponding development teams. Columns under Types represent classes causing bad-castings: allocation, source and destination classes. Columns under Security Implication represents the security impacts of each vulnerability: whether the vulnerability has C++ ABI incompatibility issues (ABI); whether the vulnerability triggers memory corruption (Mem); and the actual security assessment ratings assigned by the vendor (Rate). †: The GNU libstdc++ members did not provide security ratings; CVE-2014-1594 is assigned to Bug ID 1074280.

Product (Bug ID)	Vulnerable	Types	Security Implication		
	Function	Allocation / Source / Destination	ABI	Mem	Rate
Firefox (1074280 [10])	PaintLayer	BasicThebesLayer / Layer / BasicContainerLayer	✓	✓	High
Firefox (1089438 [11])	EvictContent	PRCListStr / PRCList / nsSHistory	✓	✓	High
libstdc++ (63345 [84])	_M_const_cast	EncodedDescriptorDatabase / Base_Ptr / Rb_Tree_node	✓	-	†
libstdc++ (63345 [84])	_M_end	EnumValueOptions / Rb_tree_node_base / Link_type	✓	-	†
libstdc++ (63345 [84])	_M_end const	GeneratorContext / Rb_tree_node_base / Link_type_const	✓	-	†
libstdc++ (63345 [84])	_M_insert_unique	WaitableEventKernel / Base_ptr / List_type	✓	-	†
libstdc++ (63345 [84])	operator*	BucketRanges / List_node_base / Node	✓	-	†
libstdc++ (63345 [84])	begin	FileOptions / Link_type / Rb_Tree_node	✓	-	†
libstdc++ (63345 [84])	begin const	std::map / Link_type / Rb_Tree_node	✓	-	†
libstdc++ (63345 [84])	end	MessageOptions / Link_type / Rb_Tree_node	✓	-	†
libstdc++ (63345 [84])	end const	Importer / Link_type / Rb_Tree_node	✓	-	†

the incorrectly casted pointer. Thus the C++ ABI and Memory columns are checked for these two cases.

CAVER also found nine bugs in GNU libstdc++ while running the Chromium browser. We reported these bugs to the upstream maintainers, and they have been confirmed and fixed. Most of these bugs were triggered when libstdc++ converted the type of an object pointing to its composite objects (e.g., Base_Ptr in libstdc++) into a more derived class (Rb_Tree_node in libstdc++), but these derived classes were not base classes of what was originally allocated (e.g., EncodedDescriptorDatabase in Chromium). Since these are generic bugs, meaning that benign C++ applications will encounter these issues even if they correctly use libstdc++ or related libraries, it is difficult to directly evaluate their security impacts without further evaluating the applications themselves.

These vulnerabilities were identified with legitimate functional test cases. Thus, we believe CAVER has great potential to find more vulnerabilities once it is utilized for more applications and test cases, as well as integrated with fuzzing infrastructures like ClusterFuzz [2] for Chromium.

Table 11: Security evaluations of CAVER with known vulnerabilities of the Chromium browser. We first picked five known bad-casting bugs and wrote test cases for each vulnerability, retaining features that may affect CAVER's detection algorithm, including class hierarchy and their compositions, and related classes including allocation, source, and destination types). CAVER correctly detected all vulnerabilities.

CVE #	Type Names			Security Rating	Mitigated
	Allocation	Source	Destination		by CAVER
CVE-2013-0912	HTMLUnknownElement	Element	SVGElement	High	✓
CVE-2013-2931	MessageEvent	Event	LocatedEvent	High	✓
CVE-2014-1731	RenderListBox	RenderBlockFlow	RenderMeter	High	✓
CVE-2014-3175	SpeechSynthesis	EventTarget	SpeechSynthesisUtterance	High	✓
CVE-2014-3175	ThrobAnimation	Animation	MultiAnimation	Medium	✓

3.6.3 Effectiveness of Bad-casting Detection

To evaluate the correctness of detecting bad-casting vulnerabilities, we tested five bad-casting exploits of Chromium on the CAVER-hardened Chromium binary (see Table 11). We backported five bad-casting vulnerabilities as unit tests while preserving important features that may affect CAVER's detection algorithm, such as class inheritances and their compositions, and allocation size. This backporting was due to the limited support for the `LLVM/clang` compiler by older Chromium (other than CVE-2013-0912). Table 11 shows our testing results on these five known bad-casting vulnerabilities. CAVER successfully detected all vulnerabilities.

In addition to real vulnerabilities, we thoroughly evaluated CAVER with test cases that we designed based on all possible combinations of bad-casting vulnerabilities: (1) whether an object is polymorphic or non-polymorphic; and (2) the three object types: allocation, source, and destination.

$$|\{\text{Poly},\ \text{non-Poly}\}|^{|\{\text{Alloc, From, To}\}|} = 8$$

Eight different unit tests were developed and evaluated as shown in 12. Since CAVER's design generally handles both polymorphic and non-polymorphic classes, CAVER successfully detected all cases. For comparison, UBSAN failed six cases mainly due to its dependency on RTTI. More severely, among the failed cases, UBSAN crashed for two cases when it tried to parse RTTI non-polymorphic class objects, showing it is difficult

Table 12: Evaluation of protection coverage against all possible combinations of bad-castings. P and Non-P mean polymorphic and non-polymorphic classes, respectively. In each cell, ✓marks a successful detection, X marks a failure, and Crash marks the program crashed. (a) and (b) show the results of CaVer with polymorphic class allocations and non-polymorphic class allocations, respectively, and (c) and (d) show the cases of UBSan. CaVer correctly detected all cases, while UBSan failed for 6 cases including 2 crashes.

(a) CaVer, P Alloc

To From	P	Non-P
P	✓	✓
Non-P	✓	✓

(b) CaVer, Non-P Alloc

To From	P	Non-P
P	✓	✓
Non-P	✓	✓

(c) UBSan, P Alloc

To From	P	Non-P
P	✓	X
Non-P	✓	X

(d) UBSan, Non-P Alloc

To From	P	Non-P
P	Crash	X
Non-P	Crash	X

Table 13: Comparisons of protection coverage between UBSan and CaVer. In the # of tables column, `VTable` shows the number of virtual function tables and `THTable` shows the number of type hierarchy tables, each of which is generated to build the program. # of verified cast shows the number `static_cast` instrumented in UBSan and CaVer, respectively. Overall, CaVer covers 241% and 199% more classes and their castings, respectively, compared to UBSan.

Name	**# of tables**		**# of verified cast**	
	RTTI	**`THTable`**	**UBSan**	**CaVer**
483.xalancbmk	881	3,402	1,378	1,967
450.soplex	39	227	0	2
Chromium	24,929	94,386	11,531	15,611
Firefox	9,907	30,303	11,596	71,930

to use without manual blacklists. Considering Firefox contains greater than 60,000 downcasts, (see Table 13), creating such a blacklist for Firefox would require massive manual engineering efforts.

3.6.4 Protection Coverage

Table 13 summarizes our evaluation of CaVer's protection coverage during instrumentation, including the number of protected types/classes (the left half), and the number of protected type castings (the right half). In our evaluation with C++ applications in SPEC CPU 2006, Firefox, and Chromium, CaVer covers 241% more types than UBSan; and protects 199% more type castings.

Table 14: The file size increase of instrumented binaries: CAVER incurs 64% and 49% less storage overheads in Chromium and Firefox browsers, compared to UBSAN.

Name	**File Size (KB)**				
	Orig.	**UBSAN**		**CAVER**	
483.xalancbmk	6,111	6,674	9%	7,169	17%
450.soplex	466	817	75%	861	84%
Chromium	249,790	612,321	145%	453,449	81%
Firefox	242,704	395,311	62%	274,254	13%

3.6.5 Instrumentation Overheads

There are several sources that increase a program's binary size (see Table 14), including (1) the inserted functions for tracing objects' type and verifying type castings, (2) the `THTable` of each class, and (3) CAVER's runtime library. Although CAVER did not perform much instrumentation for most SPEC CPU 2006 applications, the file size increase still was noticeable. This increase was caused by the statically linked runtime library (245 KB). The CAVER-hardened Chromium requires 6× more storage compared to Firefox because the Chromium code bases contains more classes than Firefox. The additional `THTable` overhead is the dominant source of file size increases. (see Table 13). For comparison, UBSAN increased the file size by 64% and 49% for Chromium and Firefox, respectively; which indicates that `THTable` is an efficient representation of type information compared to RTTI.

3.6.6 Runtime Performance Overheads

In this subsection, we measured the runtime overheads of CAVER by using SPEC CPU 2006's `C++` benchmarks and various browser benchmarks for Chromium and Firefox. For comparison, we measured runtime overheads of the original, non-instrumented version (compiled with `clang`), and the UBSAN-hardened version.

Microbenchmarks To understand the performance characteristics of CAVER-hardened applications, we first profiled micro-scaled runtime behaviors related to CAVER's operations (Table 15). For workloads, we used the built-in input for the two `C++` applications of SPEC CPU 2006, and loaded the default start page of the Chromium and Firefox browsers.

Table 15: The number of traced objects and type castings verified by CAVER in our benchmarks. Under the *Object Tracing* column, *Peak* and *Total* denote the maximum number of traced objects during program execution, and the total number of traced objects until its termination, respectively. *Global*, *Stack*, and *Heap* under the *Verified Casts* represent object's original types (allocation) involved in castings. Note that Firefox heavily allocates objects on stack, compared to Chromium. Firefox allocated 4,134% more stack objects, and performs 1,550% more type castings than Chromium.

Name	**Object Tracing**					**Verified Castings**			
	Global	**Stack**		**Heap**		**Global**	**Stack**	**Heap**	**Total**
	Total	**Peak**	**Total**	**Peak**	**Total**				
483.xalancbmk	165	32	190k	8k	88k	0	104	24k	24k
450.soplex	36	1	364	141	658	0	0	0	0
Chromium	3k	274	350k	79k	453k	963	338	150k	151k
Firefox	24k	38k	14,821k	213k	685k	41k	524k	511k	1,077k

Overall, CAVER traced considerable number of objects, especially for the browsers: 783k in Chromium, and 15,506k in Firefox.

We counted the number of *verified castings* (see Table 15), and the kinds of allocations (i.e., global, stack, or heap). In our experiment, Firefox performed 710% more castings than Chromium, which implies that the total number of verified castings and the corresponding performance overheads highly depends on the way the application is written and its usage patterns.

SPEC CPU 2006 With these application characteristics in mind, we first measured runtime performance impacts of CAVER on two SPEC CPU 2006 programs, xalancbmk and soplex. CAVER slowed down the execution of xalancbmk and soplex by 29.6% and 20.0%, respectively. CAVER-NAIVE (before applying the optimization techniques described in §3.4.4) slowed down xalancbmk and soplex by 32.7% and 20.8% respectively. For UBSAN, xalancbmk crashed because of RTTI limitations in handling non-polymorphic types, and soplex becomes 21.1% slower. Note, the runtime overheads of CAVER is highly dependent on the application characteristics (e.g., the number of downcasts performed in runtime). Thus, we measured overhead with more realistic workloads on two popular browsers, Chromium and Firefox.

Browser benchmarks (Chromium) To understand the end-to-end performance of CAVER,

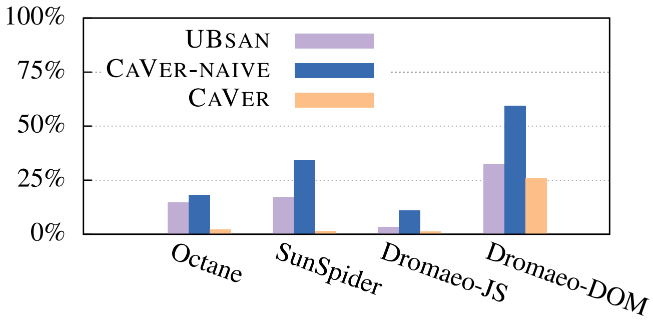


Figure 16: Browser benchmark results for the Chromium browser. On average, while CAVER-NAIVE incurs 30.7% overhead, CAVER showed 7.6% runtime overhead after the optimization. UBSAN exhibits 16.9% overhead on average.

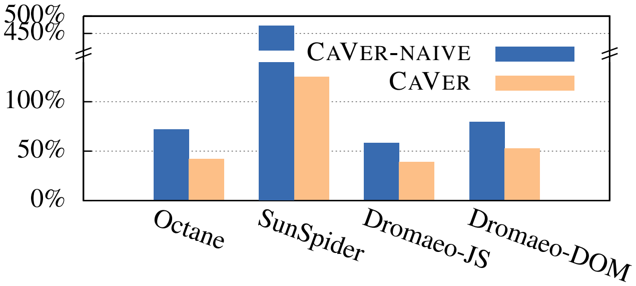


Figure 17: Browser benchmark results for the Firefox browser. On average, CAVER and CAVER-NAIVE showed 64.6% and 170.3% overhead, respectively.

we measured the performance overhead of web benchmarks. We tested four browser benchmarks: Octane [53], SunSpider [122], Dromaeo-JS [83], and Dromaeo-DOM [83], each of which evaluate either the performance of the JavaScript engine or page rendering.

Figure 16 shows the benchmark results of the Chromium browser. On average, CAVER showed 7.6% overhead while CAVER-NAIVE showed 30.7%, which implies the optimization techniques in §3.4.4 provided a 23.1% performance improvement. This performance improvement is mostly due to the safe-allocation optimization, which identified 76,381 safe-allocation types (81% of all types used for Chromium) and opted-out to instrument allocation sites on such types. Compared to UBSAN, CAVER is 13.8% faster even though it offers more wide detection coverage on type casting. Thus, we believe this result shows that CAVER's `THTable` design and optimization techniques are efficient in terms of runtime performances.

Browser benchmarks (Firefox) We applied CAVER to the Firefox browser and measured

Table 16: Runtime memory impacts (in KB) while running target programs. UBSAN crashed while running xalancbmk and Firefox due to the non-polymorhpic typed classes. *Peak* and *Avg* columns denote the maximum and average memory usages, respectively, while running the program. CAVER used 137% more memory on Chromium, and 23% more memory on Firefox. UBSAN used 158% more memory on Chromium.

Name	**Original**		**UBSAN**		**CAVER**	
	Peak	Avg	Peak	Avg	Peak	Avg
483.xalancbmk	9	8	crash	crash	14	12
450.soplex	2	2	2	2	5	5
Chromium	376	311	952	804	878	738
Firefox	165	121	crash	crash	208	157

the performance overhead for the web benchmarks used in evaluating the Chromium browser. On average, CAVER imposed 64.6% overhead while CAVER-NAIVE imposed 170.3% overhead (Figure 17). Similar to the Chromium case, most of performance improvements are from safe-allocation optimization, which identified 21,829 safe-allocation types (72% of all used types for Firefox). UBSAN was unable to run Firefox because it crashed due to the inability of its RTTI to handle non-polymorphic types, so we do not present the comparison number. Compared to CAVER's results on Chromium, the CAVER-enhanced Firefox showed worse performance, mainly due to the enormous amount of stack objects allocated by Firefox (Table 15). In order words, the potential performance impacts rely on the usage pattern of target applications, rather than the inherent overheads of CAVER's approaches.

Memory overheads UBSAN and CAVER achieve fast lookup of the metadata of a given object by using a custom memory allocator that is highly optimized for this purpose, at the cost of unnecessary memory fragmentation. In our benchmark (Table 16), UBSAN used 2.5× more memory at peak and average; and CAVER used 2.3× more memory at peak and average, which is an 8% improvement over UBSAN. Considering CAVER's main purpose is a diagnosis tool and the amount of required memory is not large ($<$ 1 GB), we believe that these memory overheads are acceptable cost in practice for the protection gained.

3.7 Discussion

Integration with fuzzing tools During our evaluations, we relied on the built-in test inputs distributed with the target programs, and did not specifically attempt to improve code coverage. Yet CAVER is capable of discovering dozens of previously unknown bad-casting bugs. In the future, we plan to integrate CAVER with fuzzing tools like the ClusterFuzz [2] infrastructure for Chromium to improve code coverage. By doing so, we expect to discover more bad-casting vulnerabilities.

Optimization In this thesis, we focused on the correctness, effectiveness, and usability of CAVER. Although we developed several techniques to improve performance, optimization is not our main focus. With more powerful optimization techniques, we believe CAVER can also be used for runtime bad-casting mitigation.

For example, one direction we are pursuing is to use static analysis to prove whether a type casting is always safe. By doing so, we can remove redundant cast verification.

Another direction is to apply alignment-based direct mapping scheme for global and stack objects as well. Please recall that red-black trees used for global and stack objects show $O(logN)$ complexity, while alignment-based direct mapping scheme guarantees $O(1)$ complexity. In order to apply alignment-based direct mapping scheme for global and stack objects together, there has to be radical semantic changes in allocating stack and global objects. This is because alignment-based direct mapping scheme requires that all objects have to be strictly aligned. This may not be difficult for global objects, but designing and implementing for stack objects would be non-trivial for the following reasons: (1) essentially this may involve a knapsack problem (i.e., given different sized stack objects in each stack frame, what are the optimal packing strategies to reduce memory uses while keeping a certain alignment rule); (2) an alignment base address for each stack frame has to be independently maintained during runtime; (3) supporting variable length arrays (allowed in ISO C99 [48]) in stack would be problematic as the packing strategy can be only determined at runtime in this case.

Furthermore, it is also possible to try even more extreme approaches to apply alignment-based direct mapping scheme—simply migrating all stack objects to be allocated in heap. However, this may result in another potential side effects in overhead.

3.8 *Related work*

Bad-casting detection The virtual function table checking in Undefined Behavior Sanitizer (UBSAN-vptr) [113] is the closest related work to CAVER. Similar to CAVER, UBSAN instruments `static_cast` at compile time, and verifies the casting at runtime. The primary difference is that UBSan relies on RTTI to retrieve the type information of an object. Thus, as we have described in §3.4, UBSAN suffers from several limitations of RTTI . (1) Coverage: UBSAN cannot handle non-polymorphic classes as there is no RTTI for these classes; (2) Ease-of-deployments: hardening large scale software products with UBSAN is non-trivial due to the coverage problem and phantom classes. As a result, UBSAN has to rely on blacklisting [30] to avoid crashes.

RTTI alternatives Noticing the difficulties in handling complex C++ class hierarchies in large-scale software, several open source projects use a custom form of RTTI. For example, the LLVM project devised a custom RTTI [76]. LLVM-style RTTI requires all classes to mark its identity once it is created (i.e., in C++ constructors) and further implement a static member function to retrieve its identity. Then, all type conversions can be done with templates that leverage this static member function implemented in every class. Because the static member function can tell the true identity of an object, theoretically, all type conversions are always correct and have no bad-casting issues. Compared to CAVER, the drawback of this approach is that it requires manual source code modification. Thus, it would be non-trivial to modify large projects like browsers to switch to this style. More alarmingly, since it relies on developers' manual modification, if developers make mistakes in implementations, bad-casting can still happen [111].

Runtime type tracing Tracing runtime type information offers several benefits, especially for debugging and profiling. [103] used RTTI to avoid complicated parsing supports in profiling parallel and scientific C++ applications. Instead of relying on RTTI, [38, 81] instruments memory allocation functions to measure complete heap memory profiles. CAVER is inspired by these runtime type tracing techniques, but it introduced the `THTable`, a unique data structure to support efficient verification of complicated type conversion.

Memory corruption prevention As described in §3.2, bad-casting can provide attackers access to memory beyond the boundary of the casted object. In this case, there will be a particular violation (e.g., memory corruptions) once it is abused to mount an attack. Such violations can be detected with existing software hardening techniques, which prevents memory corruption attacks and thus potentially stop attacks abusing bad-casting. In particular, Memcheck (Valgrind) [91] and Purify [55] are popularly used solutions to detect memory errors. AddressSanitizer [101] is another popular tool developed recently by optimizing the way to represent and probe the status of allocated memory. However, it cannot detect if the attacker accesses beyond red-zones or stressing memory allocators to abuse a quarantine zone [28]. Another direction is to enforce spatial memory safety [36, 64, 88, 89, 124], but this has drawbacks when handling bad-casting issues. For example, Cyclone [64] requires extensive code modifications; CCured [89] modifies the memory allocation model; and SVA [36] depends on a new virtual execution environment. More fundamentally, most only support `C` programs.

Overall, compared to these solutions, we believe CAVER makes a valuable contribution because it detects the root cause of one important vulnerability type: bad-casting. CAVER can provide detailed information on how a bad-casting happens. More importantly, depending on certain test cases or workloads, many tools cannot detect bad-casting if a bad-casted pointer is not actually used to violate memory safety. However, CAVER can immediately detect such latent cases if any bad-casting occurs.

Control Flow Integrity (CFI) Similar to memory corruption prevention techniques, supporting CFI [1, 125, 126, 129] may prevent attacks abusing bad-casting as many exploits hijack control flows to mount an attack. Furthermore, specific to C++ domain, SafeDispatch [62] and VTV [118] guarantee the integrity of virtual function calls to prevent hijacks over virtual function calls. First of all, soundly implementing CFI itself is challenging. Recent research papers identified security holes in most of CFI implementations [21, 40, 51, 52]. More importantly, all of these solutions are designed to only protect control-data, and thus it cannot detect any non-control data attacks [25]. For example, the recent vulnerability exploit against glibc [96] was able to completely subvert the victim's system by merely overwriting non-control data—EXIM's runtime configuration. However, because CAVER is not relying on such post-behaviors originating from bad-casting, it is agnostic to specific exploit methods.

CHAPTER IV

SIDEFINDER:SYNTHESIZING HASH TABLE TIMING-CHANNEL ATTACKS

4.1 Introduction

Traditionally, if an attacker wishes to leak or corrupt a program P's sensitive information, the attacker (1) finds a vulnerability in P that compromises the memory safety of P and then (2) provides an input to P that exploits the vulnerability to leak or corrupt information as desired, often by causing the program to execute arbitrary code on the attacker's behalf. While completely and efficiently protecting the control and data integrity of programs remains an important open problem that is the subject of much ongoing work, with many protection mechanisms, such as ASLR, DEP (data execution prevention), and CFI (control-flow integrity), deployed in commercial products [1, 71, 129], the amount of effort that an attacker must use in order to leak or corrupt information via a memory vulnerability is significantly increased.

In response to such protection mechanisms, attackers have begun to explore deeper and more challenging problems: e.g., launching attacks that do not cause a program to perform unsafe memory operations, but instead infer information about a program's sensitive data by invoking the program with selected inputs and observing public information about the execution. One such class of attacks, which infer information about sensitive data by observing program execution time, are *timing attacks* [16, 69]. Compare to the extensive body of work on finding safety vulnerabilities in programs and protecting vulnerable programs [1, 6, 19, 20, 24, 71, 90, 129], existing work on finding exploitable timing channels (i.e., vulnerabilities) is limited. Specifically, suspecting a program/algorithm to be vulnerable to timing channel is easy, but generating concrete input to verifying such hypothesis is very difficult. Existing work either (1) focuses on estimating the upper bound of leakable information [44], which may not reflect the difficulties of constructing real attacks; or (2) relies on attacker's experience (i.e., an art) to construct the input manually [7, 35, 59, 68, 70, 73, 93, 121].

In this work, we propose a technique for automatically attacking a class of new timing-channel vulnerabilities that occur in general programs that use deterministic hash tables, known as *hash-table client vulnerabilities*. Such vulnerabilities arise when an attacker who can partially control the inputs to a *hash-table client* that stores sensitive data in a hash table and who can observe the performance of the client can infer information about the sensitive data that the client stores.

In particular, if a hash-table client stores sensitive data d as a key in a deterministic hash table H, then an attacker can potentially learn information about d by **(1)** finding a sequence of program inputs I that place a sufficiently large set of entries in a target table that satisfy a set of attack-specific equality constraints, **(2)** executing the program on each input in I, and **(3)** observing the time taken to access entries in H. The key challenge in performing step **(1)** of such an attack is that even if a security analyst can identify a target hash-table that is a likely site of vulnerabilities, the analyst must be able to (1) infer the complex relationship between the inputs the analyst can provide to the client, the keys that the client stores in its hash table, and the persistent state maintained by the client; and (2) repeatedly invoke the client with a set of inputs that induce a hash-table collision. For example, in order to attack timing-channels in the inode cache that we show in the chapter, an attacker has to understand complex logic behind many different file systems code in the Linux kernel.

The key observation behind our approach is that hash-table vulnerabilities can often be discovered efficiently in practical systems code by computing *symbolic function summaries* of hash-table clients. Our approach, named SIDEFINDER, combines program slicing with symbolic execution to perform step **(1)** of the above attack schema automatically. Given a hash-table client C, threshold k of keys that must be entered in the hash table maintained by

C, and predicate P over k keys, SIDEFINDER computes a succinct, potentially-incomplete, summary R of the effect of a hash table client C on its hash table and persistent state, in terms of inputs. SIDEFINDER then derives a succinct constraint from R whose solutions model sequences of executions of C that place k keys in the hash-table maintained by C and that satisfy the predicate P. SIDEFINDER solves the derived constraint with an off-the-shelf constraint solver.

We have designed and implemented a SIDEFINDER as a tool for the LLVM intermediate language. Using SIDEFINDER, we found inputs that allowed us to exploit side-channels in the Linux kernel automatically, which otherwise required either (1) significant manual effort or (2) significantly more computational resources using a brute-force approach.
To summarize, this chapter makes the following contributions:

- We formulate timing-channel attacks on programs that store sensitive data in hash table (e.g., sensitive address information to bypass ASLR or filename to breach user's privacy) as reductions to the *hash-table client attack* problem.
- We describe a program analysis, SIDEFINDER, that synthesizes attacking inputs and thus finds solutions to the hash-table client attack problem efficiently, using symbolic function summaries.
- We present two concrete timing-channel attacks, found in the Linux kernel that we synthesized using SIDEFINDER.

4.2 Problem Scope

4.2.1 Attack Model

We assume that an attacker can execute regular operations on a target's system. In particular, the attacker can invoke any of a set of interface operations provided by the target system (e.g., runtime functions provided by the JavaScript runtime or system calls provided by the Linux Kernel), and the attacker can completely control the input provided to each operation. At least one operation can be called to insert entries into a target hash table and another which can be used to search for entries in the target hash table. The attacker can measure the time taken to execute each interface operation on an input that they provide.

The attacker's goal is to infer information about sensitive data maintained by the interface operations. While the attacker does not initially know anything about the value of the sensitive data itself, and cannot directly read or modify the addresses that store sensitive information, the attacker knows the program data structures that store the sensitive information before the attacker begins invoking operations. Among all of the attacks that we present, sensitive information is stored in different types of system memory, including (1) kernel-space address information, where the attacker is assumed to run the user-level code (§4.4.1), and (2) a privileged user's filename information, where the attacker is assumed to run code as an unprivileged user (§4.4.2). It is worth noting that (1) allows an attacker to bypass well-known security mechanism, Address Space Layout Randomization (ASLR) on the Linux Kernel, and (2) allows to breach privacy sensitive information.

4.2.2 Timing-Channel Attacks in Hash Tables

In general, a hash table is a data structure that implements an associative array, which maps a key to a value. When a hash table adds a key-value pair (k, v), the table invokes a hash function, which deterministically outputs a bucket index from k, and then stores (k, v) at the bucket index. Since it is possible for a hash table to map distinct keys to the same bucket index, a hash table employs collision resolution method [74] to handle colliding cases, such as separate chain or open addressing.

In this setting, timing-channel attacks in hash tables occur if sensitive data is used to determine a bucket index and normal inputs collide with the sensitive data. Specifically, considering a hash function h, which takes an input k to compute the bucket index, the timing-channel occurs if following two conditions hold: (1) A sensitive input (say k_s) is used to determine the bucket index with some loss of information depending on h; and (2) an attacker can find a colliding input k_c such that $h(k_c) = h(k_s)$, and k_c can be inserted to the table. Since a collision resolution process is deterministic, operations on k_c would differ depending on whether k_s exists in the table or not, which could result in timing differences. This further implies that an attacker could be able to infer whether k_s exists or not, if he/she can find multiple instances of k_c and have the hash table operate over them.

In order to launch such side-channel attacks, attackers must have sufficient control over the bucket index by only manipulating inputs of the target program. This is because only having a single instance of k_c may not introduce observable timing differences (e.g., in case of the inode cache attacks, at least 1,024 instances of k_c are required to introduce observable timing differences (§4.5.2).

However, in practice, it is very challenging to manually find such a sufficient number of colliding inputs. In commodity software, an implementation of a hash function itself is not completely isolated, but its operations are rather heavily mixed up with many other upper layer implementations. Thus, an actual hash function operation varies depending on which upper layer is in use, and thus it is challenging to understand the complete hash function operations. For example, in the dentry cache attack (§4.4.2), the input k to the hash function is not directly from the system call, but a result from another hash function (`filename->hash`), which differs across different underlying file system. Now, consider the final goal for an attacker: find a sufficient number of inputs for the given hash function, which result in the same bucket index. Manually solving this task would be challenging and time consuming given this mixed up and complicated implementations of a hash function. This would be even more difficult, considering the fact that our target software has a massive code size (e.g., the Linux Kernel).

4.3 Formulating Attacks on Hash-Table Clients

This section presents SIDEFINDER, which takes a hash-table client C, a bucket threshold k, and a predicate P over k hash-table keys; SIDEFINDER attempts to synthesize a sequence of k inputs to C that place k keys in the hash-table maintained by C that satisfy P.

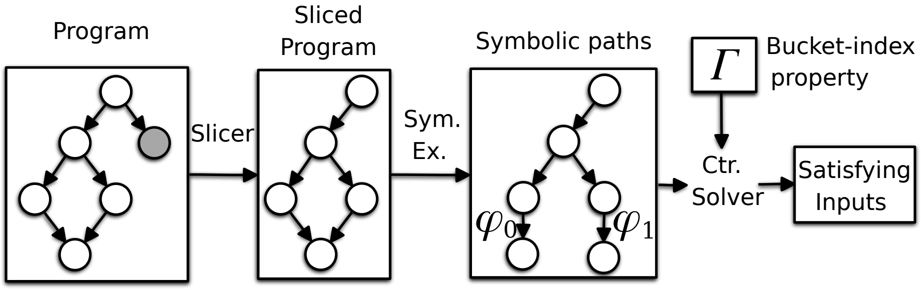


Figure 18: Workflow of the attack synthesizer, SIDEFINDER. The graphs in the figures depict the CFG of programs generated; white circles depict control locations in the backward slice from an operation that adds entries to the target hash table.

The workflow of SIDEFINDER is depicted in Figure 18: SIDEFINDER takes as input a program P, variables `key` and `buk` that store the key and bucket index when entries are added to the table maintained by P, and property γ over bucket values, and performs the following steps. ① SIDEFINDER first performs a backwards slice of P for `key` and `buk` to obtain a program slice P' that contains only program paths that may affect the values stored in `key` and `buk`. ② SIDEFINDER then runs a symbolic execution engine on P' to obtain a set of symbolic paths S that represent runs of P'. ③ SIDEFINDER then uses S to construct a constraint in the theory of bitvectors such that each solution of φ describes a sequence of program inputs that satisfy γ. ④ SIDEFINDER then attempts to synthesize a solution to φ by invoking an off-the-shelf constraint solver, and returns the result provided by the constraint solver.

4.3.1 Problem Definition

In this section, we formulate the problem of synthesizing an attack on a hash-table client. We first formulate a simple language of reactive programs that maintain persistent state. Let there be fixed spaces of *temporary variables* `Tmp`, *persistent variables* `Perm`, control locations `Locs`, and instructions Instrs. The union of temporary and permanent variables is denoted $\texttt{Vars} = \texttt{Tmp} \cup \texttt{Perm}$. A program consists of (1) an initial control location, (2) a final control location, and (3) a set of control edges. I.e., the space of programs is $\mathcal{P}(\texttt{Locs} \times \mathsf{Instrs} \times \texttt{Locs})$.

A program defines a language of sessions with its environment, in which each session is a sequence of program runs. Let the space of *machine words* be denoted W. Let a *temporary store* be an evaluation of temporary variables (i.e., $\Sigma_T = \texttt{Tmp} \rightarrow W$), and let a *permanent store* be an evaluation of permanent variables (i.e., $\Sigma_P = \texttt{Perm} \rightarrow W$). A *store* is a pair of a temporary store and a permanent store; i.e., the space of stores is denoted $\Sigma = \Sigma_T \times \Sigma_P$. Each program instruction $\texttt{i} \in \mathsf{Instrs}$ defines a transition relation over stores $\rho(i) \subseteq \Sigma \times \Sigma$.

A program *state* is a control location, temporary store, and permanent store; i.e., the space of states is denoted $\mathsf{States} = \texttt{Locs} \times \Sigma_T \times \Sigma_P$. A *run* of a program $P = (\texttt{i}, \texttt{f}, E)$ is a sequence of states $r = (\texttt{L}_0, \sigma_0), \ldots, (\texttt{L}_n, \sigma_n)$ such that **(1)** the control location of the initial state is the initial control location of P (i.e., $\texttt{L}_0 = \texttt{i}$); **(2)** the control location of the final state is the final control location of P (i.e., $\texttt{L}_n = \texttt{f}$); **(3)** all adjacent states in r are allowed by the control edges of P and transition relation of the instruction connecting their control edges. I.e., for each $0 \leq j < n$, $(\texttt{L}_j, \texttt{i}, \texttt{L}_{j+1}) \in E$ and $\sigma_j, \sigma_{j+1} \in \rho(\texttt{i})$. A *session* is a sequence of runs in which the permanent stores in the final state of each run is the initial permanent store of the subsequent run in the session. I.e., a sequence of runs $R = r_0, \ldots, r_k$ is a session if for each $0 \leq j < k$, $r_j = q_j^0, \ldots, (\texttt{f}, (t'_j, p'_j))$, and $r_{j+1} = (\texttt{i}, (t_{j+1}^0, p_{j+1}^0)), \ldots, q_{j+1}^n$, $p'_j = p_{j+1}^0$.

The problem of synthesizing an attack on a hash table client is, given designated variables that store the key added to a hash table by its client and the bucket at which the key is added, a bucket threshold k and a desired predicate φ on hash-table bucket values, to synthesize initial temporary stores for k runs that generate k distinct keys entered at buckets that satisfy a φ.

Definition 1 *An instance of the* HTCA *problem is a five-tuple* $(P, \texttt{key}, \texttt{buk}, k, \Gamma)$ *where:* ***(1)*** *P is a program with initial control location* $\texttt{L}_0$*;* ***(2)*** $\texttt{key}$ *stores the key of the entry committed to the hash table at the end of each run;* ***(3)*** $\texttt{buk}$ *stores the bucket value at which each key is added at the end of each run;* ***(4)*** *$k \in \mathbb{N}$ is a minimum required* bucket threshold *of distinct keys;* ***(5)*** *for* Buks_k *a vector of k copies of* $\texttt{buk}$*, Γ is a Boolean combination of equality* *constraints over integer constants and* buk*.*

A solution *to the* HTCA *problem* $(P, \texttt{key}, \texttt{buk}, k, \Gamma)$ *is a set of* k *temporary stores* Σ_T^* *such that there is some session* $S = r_0, \ldots, r_{k-1}$ *in which:*

1. *For each* $0 \leq j < k$ *and run* $r_j = (\texttt{i}, (t_j^0, p_j^0))$*,* t_j^0 *is the* j*th store in* Σ_T^**.*

2. *For all* $0 \leq m \neq n < k$ *and runs* $r_m = q_m^0, \ldots, (\texttt{f}, (t_m, p_m))$ *and* $r_n = q_n^0, \ldots, (\texttt{f}, (t_n, p_n))$*,* $p_m(\texttt{key}) \neq p_n(\texttt{key})$*.*

3. *For each* $0 \leq n < k$ *with* $r_n = q_n^0, \ldots, (\texttt{f}, (t_m, p_n))$*, the interpretation* $\iota : \mathsf{Buks} \to W$ *defined as* $\iota(\texttt{buk}_j) = p_j(\texttt{buk}_j)$ *satisfies* Γ*.*

4.3.2 Synthesizing Attacks on Hash-Table Clients

In this section, we present our solver, SIDEFINDER, for the HTCA problem. We first describe the components of SIDEFINDER, in particular its program slicer (§4.3.2.1), constraint solver (§4.3.2.2), and symbolic-execution engine (§4.3.2.3). We then describe how SIDEFINDER uses each of these components to attempt to solve the HTCA problem (§4.3.2.4).

4.3.2.1 Program Slicer

SIDEFINDER uses a *backwards program slicer* SLICER, which takes as input a program P, a set of program variables X, and a control location L, and generates a program P′ such that (1) the variables of P′ are all variables of P that may affect the value of the variables in X when a run reaches L; (2) for each run r of P that reaches L, there is a run r' of P′ with equal values in all variables in X. Program slicing is a well-studied problem in program analysis and transformation [57, 123].

However, as is the case with most program analyses, to implement an analysis that can be run on practical systems software, the analysis writer must carefully balance the precision of the results against scalability. In order to analyze kernel functions at scale, SLICER uses a dependency analysis that is flow-insensitive and context-sensitive. However, because we found that in practice, kernel code contains many loads and stores that may read and write from structures of multiple possible types, the dependency analysis is field sensitive. For callsite c that may invoke a call target indirectly, SLICER's dependency analysis assumes that the program may call any function at c.

4.3.2.2 A Constraint Solver for the Theory of Bitvectors

SIDEFINDER represents sets of program states as formulas in a the theory of bitvectors. A formula in the theory of bitvectors represents a set of (satisfying) assignments from logical variables to bitvectors of fixed width. Terms in the theory of bitvectors model common program operations (e.g., bounded arithmetic and bitwise-logical operations); predicates in the theory model common program tests of bitvectors values (e.g., equalities and inequalities). Formulas in the theory of bitvectors are commonly used in program analyses to efficiently represent sets of program states [20].

Definition 2 *Let the theory of bitvectors be denoted* BV*, and let the spaces of* BV *terms and formulas over variables* X *be denoted* $T_{\mathrm{BV}}[X]$ *and* $F_{\mathrm{BV}}[X]$*, respectively. For vectors of variables* X *and* Y *of equal length, let* $\mathsf{inst}[\varphi, Y]X$ *denote the formula* φ *with each variable* $x_i \in X$ *replaced with its corresponding* i*th variable* $y_i \in Y$*.*

The bitvector-constraint solver SOLVEBV *takes as input a formula* $\varphi \in F_{\mathrm{BV}}[X]$ *and outputs one of the following values:* ***(1)*** *If* φ *is satisfiable, then* SOLVEBV *returns a satisfying assignment of each variable in* X *to a machine word.* ***(2)*** *If* φ *is unsatisfiable, then* SOLVEBV *returns* Unsat.

Previous work has developed implementations of multiple efficient constraint solvers for BV. [41]. SIDEFINDER uses the STP theorem prover [47].

4.3.2.3 A Symbolic-Execution Engine

SIDEFINDER uses a symbolic-execution engine SYMEX. SYMEX takes as input a program P and returns a set of symbolic descriptions of the computation performed along full paths of P.

```
input  : An HTCA problem $H = (P, \texttt{key}, \texttt{buk}, k, \Gamma)$.
output : Either (1) a solution to $H$ or (2) None.
$P' := \textsc{Slicer}(P, \texttt{f}^P, \{\texttt{key}, \texttt{buk}\})$ ;
$S := \textsc{SymEx}(P')$ ;
$\varphi := \textsc{ConsHyper}(S, k, \Gamma)$ ;
switch SolveBv($\varphi$) do
    case $m$   return $m|_{V^k}$ ;
    case Unsat   return None ;
endsw
```

Algorithm 1: SideFinder: a solver for the HTCA problem.

Definition 3 *Let a* symbolic path (φ, σ) *be a pair of a formula* $\varphi \in F_{\textsc{Bv}}[\texttt{Vars}]$ *and a map* $\sigma : \texttt{Vars}' \to T_{\textsc{Bv}}[\texttt{Vars}]$. *The* symbolic-execution engine SymEx *takes as input a program* P *and outputs a set of* n *symbolic paths* $\{(\varphi_0, \sigma_0), \ldots, (\varphi_n, \sigma_n)\}$. *For each* $0 \leq i \leq n$, *valuation of pre-state variables* $\sigma : \texttt{Vars} \to W$, *and valuation of post-state variables* $\sigma' : \texttt{Vars}' \to W$, *if (1)* σ *satisfies* φ_i *and (2) for each post-state variable* $\texttt{x}' \in \texttt{Vars}'$ *such that* $\sigma'(\texttt{x}')$ *equals the valuation of* $\sigma(\texttt{x}')$ *on* V, *there is a run of* P *with initial store* σ *and final state* σ'.

SideFinder uses S2E [26], which is based on Klee [19], as its implementation of SymEx. In §4.3.3, we describe how we address practical issues that arise when using Klee to symbolically execute kernel functions.

Each set of symbolic paths S defines the transition relation over all control paths in S, denoted $\mu_S \in \textsc{Bv}[\texttt{Vars}, \texttt{Vars}']$. The construction of μ_S is standard, and we omit a detailed description.

4.3.2.4 A Solver for HTCA

SideFinder uses the program slicer Slicer, Bv constraint solver SolveBv, and symbolic-execution engine SymEx to solve HTCA instance $H = (P, \texttt{key}, \texttt{buk}, k, \Gamma)$ (shown in **Algorithm 1**). SideFinder first invokes the program slicer Slicer on input program P, the final location of P, and program variables key and buk to obtain the program slice P' SideFinder then invokes the symbolic-execution engine SymEx on P' and target control location L to obtain a set of symbolic paths S. SideFinder then invokes a procedure

ConsHyper on symbolic paths S, the bucket threshold k, and bucket constraint Γ to obtain a Bv formula φ such that each satisfying assignment of φ is a solution to H; we describe the implementation of ConsHyper in detail below. SideFinder then invokes the Bv constraint solver SolveBV on φ to attempt to find a satisfying assignment of φ. If SolveBV finds such an assignment m, then SideFinder returns m restricted to the pre-state variables V^k as a solution to the given Htca problem. Otherwise, if SolveBV determines that φ has no satisfying assignment, then SideFinder returns that it has found no solution to H.

Implementation of ConsHyper Given a set of symbolic paths S, bucket threshold k, and bucket constraint Γ, ConsHyper returns a Bv constraint φ such that each satisfying assignment of φ contains a solution of the Htca problem given to SideFinder. The variables of φ consist of k distinct copies of `Perm` and k distinct copies of unprimed and primed copies of `Tmp`. We denote the ith copy of `Perm` as $\texttt{Perm}_i$, and we denote the unprimed and primed copies of `Tmp` as $\texttt{Tmp}_i$ and $\texttt{Tmp}'$, respectively.

An assignment $\iota : \texttt{Vars}^* \to W$ satisfies φ if: **(1)** ι satisfies the symbolic under-approximation of P defined by S, instantiated on each copy of unprimed and primed variables; **(2)** ι maps each primed key to a distinct value; **(3)** ι satisfies Γ with each copy of buk replaced with its primed copy. I.e., ConsHyper returns the following Bv constraint:

$$\begin{aligned} &\textbf{(1)} \bigwedge_{0 \leq i < k} \mu_S[\texttt{Tmp}_i, \texttt{Perm}_i, \texttt{Tmp}'_i\texttt{Perm}_{i+1} / \texttt{Vars}, \texttt{Vars}'] \\ \land\ &\textbf{(2)} \bigwedge_{0 \leq i \neq j < k} \texttt{key}'_i \neq \texttt{key}'_j \\ \land\ &\textbf{(3)}\ \Gamma[\texttt{buk}'_0/\texttt{buk}_0, \ldots, \texttt{buk}'_{k-1}/\texttt{buk}_{k-1}] \end{aligned}$$

4.3.3 Solver Properties

In this section, we consider the properties of SideFinder as a solver for Htca. We say that SideFinder is *sound* if whenever SideFinder determines that a Htca problem H

has a solution Σ, then Σ is a valid solution to H; we consider SIDEFINDER to be *complete* if whenever H has a solution, then SIDEFINDER returns a solution.

4.3.3.1 Limitations to Soundness

The soundness of SIDEFINDER is limited by the ability of SYMEX to accurately model practical reactive programs. Most implementations of SYMEX (including KLEE) typically cannot be applied directly to the systems code that we analyze due to differences between the assumptions that SYMEX places on the execution of programs compared to the actual execution model of systems code. In particular, SYMEX assumes that it is invoked to symbolically execute a program at the beginning of the program's execution; thus, all program variables to store memory objects according to the program's initialization code. However, we typically wish to apply SYMEX to analyze code that implements system calls, which may be re-executed multiple times, and maintain state in global variables that satisfy critical invariants.

Thus SIDEFINDER is typically invoked multiple times by a program tester to *lazily* introduce invariants on global data structures. I.e., a program tester first runs SIDEFINDER in an *underconstrained* context, in which no constraints are placed on the data in global memory. If SIDEFINDER does not find any solutions in such a context, then the tester does not further attempt to use SIDEFINDER to find solutions. Otherwise, if SIDEFINDER finds a solution I, then the tester manually determines if each input in I satisfies known invariants for system global data structures. If so, then the tester uses I to proceed with their attack. Otherwise, the tester determines an invariant φ over global data structures that prohibits some input in I, and directs SYMEX to assume that φ holds when the system call is entered (in particular, KLEE supports a `klee_assume` directive that allows a user to specify conditions that KLEE may assume to hold at an annotated control location).

SIDEFINDER would require significantly less effort to use if it could infer likely invariants for global data-structures and analyze system calls under inferred invariants. The problem of automatically inferring *harnesses* (i.e., invariants on persistent data) under which to analyze reactive programs remains a critical and open problem in the literature on program-analysis [65].

4.3.3.2 Limitations to Completeness

The *completeness* of SIDEFINDER is limited by the set of symbolic paths collected by SYMEX that model runs of a subject program. While we presented SIDEFINDER as modeling sets of program states as well as formulas in theory of fixed-width bitvectors, the full implementation of SIDEFINDER actually uses the theory of bitvectors *with arrays*, which allows it to accurately model the complete memory of a program state [20]. If SYMEX returns a set of symbolic paths S of the sliced program P' such that each actual program path to the target control location satisfies some symbolic path in S, then SIDEFINDER is complete. However, in general, it may not even be possible to represent all runs of P' with a finite set of symbolic paths, in particular if P' contains a loop that may be executed an unbounded number of times. However, if P' contains only loops that execute a fixed number of times independent of the input, then practical implementations will return a finite set of symbolic paths. SIDEFINDER is thus a complete solver in this restricted case.

4.4 Attacks Reduced to HTCA

In this section, we present two timing-channel attacks in security-critical uses of hash tables. For each of attack, we also show how it can be reduced to an instance of HTCA. Table 17 provides a summary of each attack.

4.4.1 Inode Cache Attacks

Inode cache. A `inode` object is abstract representations of a file. Because searching for a file is a performance-critical operation, the Kernel maintains a global cache `inode_hashtable` for `inode` objects. Using `inode_hashtable`, the Kernel does not need to access sluggish low-level file-system disks to locate an `inode` if its corresponding file is

Table 17: Summary of timing-channel vulnerabilities that we have found in security-critical programs that use hash tables. Each entry contains the type of security-critical platform (**Target**), the data structure that implements the hash table (**DS**), the program functions used to hash keys (**Hash**), insert entries (**Insert**), lookup entries (**Lookup**), the mathematical hash function implemented (**Hash function**), and used data structure (a_1, a_2) that might leak through the timing channel.

Target	DS	Hash	Insert	Lookup	Hash function	a_1	a_2	Leak
Kernel	`dentry`	`d_hash()`	`d_rehash()`	`d_lookup()`	$t = (i_1 + i_2)$ $i = (t + t \gg c_1)\&c_3$	`parent`	`ino`	File name
Kernel	`inode`	`inode_hash()`	`insert_inode_locked()`	`iget_locked()`	$t = (a_1 \times a_2) \wedge ((c_1 + a_1)/c_2)$ $i = t \oplus ((t \wedge c_1) \gg c_3)$	`sb`	`ino`	`sb`'s address

```
// @fs/inode.c (hash function)
// GOLDEN_RATIO_PRIME & L1_CACHE_BYTES are constant values.
// i_hash_shift & i_hash_mask are global variables,
unsigned inode_hash(struct super_block *sb, unsigned long h) {
  unsigned tmp = (h * (unsigned long)sb);
  tmp = tmp ^ (GOLDEN_RATIO_PRIME + h) / L1_CACHE_BYTES;
  tmp = tmp ^ ((tmp ^ GOLDEN_RATIO_PRIME) >> i_hash_shift);
  return tmp & i_hash_mask;
}
// @fs/inode.c (search inode from a hash table)
struct inode *iget_locked(struct super_block *sb,
                          unsigned long h) {
  unsigned bucket_index = inode_hash(sb, h);
  struct hlist_head *head = inode_hashtable + bucket_index;
  struct inode *inode = NULL;
  // Iterate each entry
  hlist_for_each_entry(inode, head) {
     // Found the corresponding inode object.
     // ...
  }
  // Failed to find.
  return NULL;
}
// @fs/inode.c (insert inode to a hash table)
void insert_inode_hash(struct inode *inode, unsigned long h) {
  struct super_block *sb = inode->i_sb;
    struct hlist_head *b = inode_hashtable + inode_hash(sb, h);
    hlist_add_head(&inode->i_hash, b);
}
```

Figure 19: A simplified excerpt of code that maintains `inode_hashtable`. `inode_hash` computes a bucket index from a given superblock address and hash value, which is usually the `inode` number. `insert_inode_hash` adds a given `inode` with a given hash value to the `inode` cache. `iget_locked` searches for a given hash value in a superblock.

frequently accessed. `inode_hashtable` resolves hash collisions using a separate chain with a linked list.

The `open()` system call uses `inode_hashtable` to optimize performance of opening a file at filename `f` (Figure 19). To add an `inode` stored in variable `inode` with an `inode` identifier stored in variable `h` (depending on the file system, this can be the `inode` number, or a preprocessing result), `open` calls the function `insert_inode_hash` on `inode` and `h`.

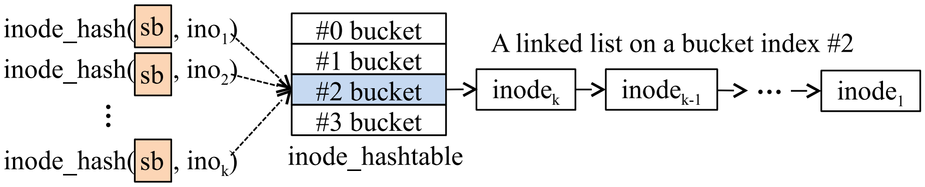


Figure 20: A depiction of `inode_hashtable` state during the attack described in §4.4.1, in the case that the chosen superblock value is correct. `inode_hashtable` hashes each key to the same bucket (2), forming a list in only bucket 2 whose length is proportional to the number of keys inserted.

`insert_inode_hash` hashes the superblock of `inode` and `h`, to obtain the head `b` of the corresponding bucket list, and adds `inode` as the new head of its bucket list. To find an `inode` with identifier `h` in superblock `sb`, `open` calls `iget_locked` on `sb` and `h`. `iget_locked` computes the bucket index `bucket_index` of `sb` and `h` by invoking the hash function `inode_hash`. Finally, `iget_locked` iterates over the linked list stored at `bucket_index` until it finds the given `inode` object.

A timing channel. The address of the superblock, stored in `sb`, is sensitive information: in particular, if an attacker could infer the address, then they could use the address to break Kernel ASLR. Although `open` does not explicitly leak information about `sb`, an attacker can infer `sb` through a timing channel in the code that maintains `inode_hashtable` by inserting a sufficiently large number of `inode` in chosen buckets of `inode_hashtable`.

Specifically, the attacker can infer information about `sb` by performing the following test T_I on hypothetical address values. (1) The attacker chooses a *timing threshold* k, which is a number of table entries that will observably affect the performance of the `inode` table, as described below. (2) The attacker chooses a hypothetical value $\texttt{sb}_H$ for the superblock address. (3) The attacker finds a set of distinct inode numbers $\{x_1, \ldots, x_k\}$ such that `inode_hash(`$\texttt{sb}_H$`, ` x_i`)` = `inode_hash(`$\texttt{sb}_H$`, ` x_j`)` for all $i, j \leq k$; (4) The attacker creates or finds[1] a set of files $\{F_1, \ldots, F_k\}$, where each file F_i has the corresponding inode number x_i for $0 \leq i \leq k$. (5) The attacker calls `open` on the sequence of files $F_1, F_2, \ldots, F_k$, populating the `inode` table with corresponding inodes $\{i_1, \ldots, i_k\}$. All inodes in I will be in the same bucket, and thus will be stored as a linked list, with i_k as the head and i_1 as the

[1] `inode` number is accessible via `ls -i` or `stat`.

tail. (6) The attacker calls `open` on the sequence of files $F_k, F_{k-1}, \ldots, F_1$, and measures the execution time for each call. (7) The attacker decides that $\texttt{sb}_H$ is the value of the superblock if and only if the execution time for each call to `open` increased.

Figure 20 depicts the state of the `inode` cache if the hypothetical superblock value $\texttt{sb}_H$ is correct (i.e., $\texttt{sb}_H$ = `sb`) after adding `inode` for the sequence of created files $F_1, \ldots, F_k$. Because $\texttt{inode\_hash(}sb, x_i\texttt{)}$ will have the same value for all $0 \leq i \leq k$, opening the files, $F_1, \ldots, F_k$, will insert corresponding `inode` into the same bucket index (#2). Moreover, because `hlist_add_head` adds each element as the new head of its bucket list, the `inode` of F_k is the head of the bucket list, while the `inode` of F_1 is the tail. As a result, if $1 \leq i < j \leq k$, search for file F_j will take less time than searching for file F_i. However, if the hypothetical superblock value is incorrect (i.e., $\texttt{sb}_H \neq \texttt{sb}$), the `inode` for $F_1, \ldots, F_k$ will not necessarily be inserted into the list for the same bucket. Thus, there may exist $i < j$ for which searching for F_i takes less time than search for F_j.

Security Implications. Based on the timing channel attack, the superblock address is now known to the attacker. This unexpected information leakage can severely weaken the protection level of the Linux kernel. For example, an attacker does not need an additional information leakage vulnerability to launch the privilege escalation attacks. In many Linux distributions, code addresses are fixed in the specific version due to the strict constraints on the kernel memory layouts, but data addresses are not as it is allocated dynamically from the large memory pool. Thus, the attacker can locate the function pointer inside the superblock structure (which is unknown to the attacker as it is data addresses), and overwrite the function pointer to mount control-flow hijacking attacks. Considering CVE-2013-6282 as an specific example, which offers arbitrary memory writes to the kernel memory spaces, the attacker can overwrite the function pointer in the superblock structure without additional leakage capability.

Reducing to HTCA Recall that in the attack on the `inode` cache as described in §4.4.1, `insert_inode_hash` is the function in which entries are added to the `inode` table, `L28` is

```
// d_hash() returns a bucket in a hash table,
struct hlist_bl_head *d_hash(const struct dentry *parent,
                             unsigned hash) {
  unsigned tmp = hash + (unsigned)parent;
  tmp = tmp + (tmp >> d_hash_shift);
  unsigned index = (tmp & d_hash_mask);
  return dentry_hashtable + index;
}
// Insert a given file and filename into a parent directory.
void __d_insert(struct dentry *parent, struct dentry* child) {
  unsigned bucket_index = d_hash(parent, child->name->hash);
  struct hlist_bl_head *head = dentry + bucket_index;
  hlist_add_head(&child, head);
  return;
}
// Search for filename in the parent dentry.
struct dentry *__d_lookup(struct dentry *parent,
                          struct qstr *filename) {
  // hlist_bl_head is the head node of a linked list.
  struct hlist_bl_head *b = d_hash(parent, filename->hash);
  hlist_bl_for_each_entry_rcu(dentry, node, b) {
    // Walks a linked list on the corresponding bucket.
    // ...
  }
}
```

Figure 21: A simplified excerpt of the code that maintains the `dentry_hashtable`. `d_hash` computes a hash from a given parent directory and the hash of a child. `d_insert` inserts an entry. `__d_lookup` finds the child of a given parent directory at a given filename.

the control location at which entries are added, `inode` is the variable the stores the key to be added, `b` stores the bucket index of the key, and `sb` stores the address of the superblock, which the attack attempts to infer. For each timing threshold $k \in \mathbb{N}$ and address value $\texttt{sb}_H \in W$, let $\textsc{Htca}_i(k, \texttt{sb}_H) = (\texttt{open}, \texttt{inode}, \texttt{b}, k, \Gamma_{k,\texttt{sb}_H})$, where $\Gamma_{\texttt{sb}_H,k}$ constrains that (1) the bucket values at the end of each run are equal and (2) in each run, the superblock variable `sb` stores address $\texttt{sb}_H$.

An attacker can perform step (4) of the hash-table attack on the `inode` table presented in §4.4.1 by solving the $\textsc{Htca}$ problem $\textsc{Htca}_i(k, \texttt{sb}_H)$, where $\texttt{sb}_H$ are chosen timing threshold and hypothesis superblock address chosen in steps (2) and (3).

4.4.2 Dentry Cache Attacks

Dentry cache. A `dentry` is an abstract representation of a directory, and the Linux Kernel maintains a global cache `dentry_hashtable` for `dentry` objects. To traverse a directory structure in a file system, the Kernel first splits an absolute file path into sub-paths on separator strings (i.e., '/'), and then tries to locate a `dentry` object starting from the first

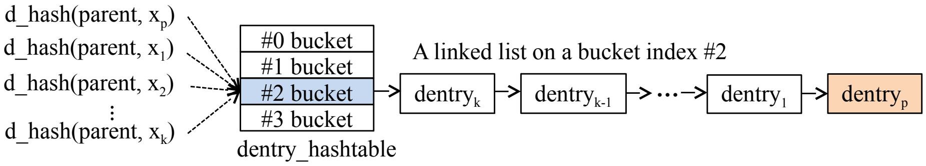


Figure 22: A depiction of `dentry_hashtable` state during the attack described in §4.4.2, in the case that the chosen filename is correct. `dentry_hashtable` hashes each key to the same bucket (2), forming a list in bucket 2 whose length is proportional to the number of keys inserted.

sub-path after root. Figure 21 contains code that maintains the cache `dentry_hashtable`. `__d_lookup()` looks up a `dentry` object in `dentry_hashtable` from its parent `dentry` object (`parent`) and name of this sub-path (`filename`). Please note that the input to the `d_hash()` function is not the string of `filename`, but a preprocessed hash value from the string.

A timing channel. `dentry_hashtable` has a timing channel through which an attacker can infer sensitive information. In particular, the attacker can decide if a target file f that can only be accessed or listed by a privileged user u (e.g., root) has filename x_p. We also assume that the attacker can induce u to access f (without directly revealing to the attacker the name of f); in practice, this assumption may be satisfied, e.g., by invoking a daemon that executes on behalf of u. For example, setuid-bit enabled programs, including `chsh` and `dotlockfile`, opens a configuration file specified by non-privileged users.

To carry out the attack, the attacker performs the following test, denoted T_D. (1) The attacker induces u to access f, and measures the amount of time t taken to perform the access. (2) The attacker chooses a *timing threshold* k, which is a number of table entries that will observably affect the performance of the `dentry` cache, as described below. (3) The attacker induces u to access a sequence of filenames $F = x_1, x_2, \ldots, x_k$ such that for each $0 \leq i \leq k$, `d_hash(parent,` x_p`)` $=$ `d_hash(parent,` x_i`)`. The value of parent does not need to be known to satisfy this condition, because a sequence of filenames in F will be stored in the same directory and thus the same value of parent will be used. The k objects will be stored as a linked list in the same bucket, with the entry for x_k as the head and the entry for x_1 as the tail. (4) The attacker induces u to access f again, and measures the amount of time t' taken to perform the access. (5) The attacker decides that f has filename

x_p if t' is significantly larger than t. Figure 22 depicts the state of dentry_hashtable if x_p is the actual file name of f. In this case, searching for f would take a long amount of time because other dentry objects will be inserted in the linked list at the same bucket index as x_p. However, in the other case, where x_p is not the actual file name of f, searching for f in may not take a long time.

Security Implications. The unprivileged attacker can leak the filename of the privileged user. Taking the dotlockfile as an example, the attacker was not able to directly learn whether a certain file exists under its setuid user, mail. With our dentry cache attack, this privacy leakage now becomes possible.

Reducing to HTCA A critical step of the attack on the dentry table can be reduced to HTCA. Recall that in the code that manipulates the dentry cache, L13 is the location at which the dentry cache is updated, child stores the key added to the dentry cache, and bucket_index stores the bucket in which the entry is added. For queried filename $\texttt{x}_\texttt{p} \in W$, address of the parent directory $\texttt{dpar} \in W$, and bucket threshold $k \in \mathbb{N}$, let $\textrm{HTCA}_d(\texttt{x}_\texttt{p}, \texttt{dpar}, k) = (\texttt{open}, \texttt{child}, \texttt{bucket\_index}, k, \Gamma_{\texttt{x}_\texttt{p},\texttt{dpar},k})$, where $\Gamma_{\texttt{x}_\texttt{p},\texttt{dpar},k}$ constrains that (1) the filename key in the 0th run is $\texttt{x}_\texttt{p}$, (2) the parent directory in each run is dpar, and (3) the bucket indices at the end of each run are equal. I.e., $\Gamma_{\texttt{x}_\texttt{p},\texttt{dpar},k}$ contains the constraint $\texttt{x}_\texttt{p} = \texttt{child}_0$; for each $0 \leq i < k$, a constraint $\texttt{parent} = \texttt{dpar}$; and for each $0 \leq i < k - 1$, a constraint $\texttt{b}_i = \texttt{b}_{i+1}$.

An attacker can reduce step (2) of the attack on the dentry cache described in §4.4.2 to solving the HTCA problem $\textrm{HTCA}_d(f, \texttt{dpar}, k)$, where f is the query filename, dpar is the address of the parent directory, and k is the bucket threshold chosen in step (1).

4.5 Implementation

The implementation on SIDEFINDER largely includes two components: a program slicer and a symbolic execution engine. A program slicer is implemented based on LLVM 3.7, and we have added 4,664 lines of C++ code. We implemented the backward-dataflow analysis in a program slicer to be flow-insensitive, context-sensitive, and field-sensitive. To analyze the Linux kernel (v4.6-rc7), we applied several minor patches of the LLVMLinux project [77] to build the kernel with LLVM. Instead of directly analyzing a single linked IR for the kernel, our analysis takes the iterative framework on multiple bitcode files to efficiently handle them.

To implement concolic execution engine, we used S2E [26], which in turn based on the symbolic execution engine KLEE [19]. In total, we added 116 lines of code in `C++`. The key change is in adding a special op code for a helper function, `get_examples(void *address, size_t size,` which finds unique `num` concrete values in the location `address`.

4.5.1 Effectiveness of SideFinder

To evaluate the effectiveness of SideFinder, this section answers the following questions:

- How well does program slicing of SideFinder discover many different uses of hash functions?

- How effective SideFinder is in synthesizing a sufficient number of timing-attack inputs?

Program slicing. The program slicing results are shown in 18: FS column shows the file system name associated with the row; **#S** column shows the number of identified taint sources by SideFinder; **True src. description** column shows a variable and function name information on the true taint source, which truly leads to the bucket index computation of a corresponding hash function in response to `open()` syscall. We particularly focused on this open() syscall as we will launch actual attacks by invoking this; **I** shows whether SideFinder was actually able to identify the true taint source.

In the case of inode cache, we instructed SideFinder to perform a backward slicing from the second parameter of `inode_hash()` function as shown in Figure 19. The slicing outputs different results for each file system, as these results are relying on each file system's

Table 18: Program slicing results of SideFinder on inode cache and dentry cache. **#S**: the number of identified taint sources; **True src. description**: a variable and function name information on the true taint source, which truly leads to the bucket index computation of a corresponding hash function; **I**: whether SideFinder was able to identify the true taint source.

(a) Inode cache

FS	#S	True src. description	I
`ext2`	1	`inode@ext2_inode_by_name()`	✓
`ext4`	11	`inode@ext4_lookup()`	✓
`reiserfs`	5	`k_objectid@reiserfs_iget()`	✓
`jfs`	3	`inum@jfs_lookup()`	✓
`btrfs`	4	`objectid@btrfs_iget_locked()`	✓
`xfs`	0	N/A	N/A

(b) Dentry cache

FS	#S	True src. description	I
`ext2`	2	`filename` in `do_sys_open()`	✓
`ext4`	5		✓
`reiserfs`	6		✓
`jfs`	3		✓
`btrfs`	7		✓
`xfs`	2		✓

implementation characteristics. For `ext2`, SideFinder precisely identified the true source, which is the inode number in the function `ext2_inode_by_name()`; For `ext4`, eleven sources were identified, and the true source (i.e., `inode` were included among these. In the case of `reiserfs`, among five sources identified, the true source `k_objectid` were also included.

It is worth noting that these identified location of the true source are not an interface function that adversaries can directly access and provide a manipulate input (i.e., `open()` syscall). These sources were mostly maintained in the data structure that is heavily aliased with other data structures associated with many other code locations. For this reason, SideFinder misses such information as the current version of SideFinder lacks of a sophisticated alias analysis.

Interestingly, `xfs` does not yield any taint sources. We manually analyzed `xfs` and confirmed that `xfs` does not use inode cache and maintains its own cache. This missing is an expected result of SideFinder (i.e., SideFinder requires users to specify the point where the hash function computation is performed) and it shows one nice property of our

Table 19: Results on synthesizing 2,048 colliding inputs using concolic execution. **Path**: The number of symbolic path constraints from the symbolization point to the bucket index computation point in a hash function; **Bucket**: The number of symbolic constraints specific to the bucket index variable (Note, there is only one constraint for the bucket); **# OP**: The total number of operations in all constraints either in path or bucket; **Time**: A solving time taken to obtain 2,048 colliding inputs; **Collide?**: check whether synthesized inputs truly collide in real executions.

(a) Inode cache

FS	**Path**		**Bucket**	**Time (s)**	**Collide?**
	#	# OP	# OP		
`ext2`	3	112	48	319	✓
`ext4`	4	132	50	336	✓
`reiserfs`	2	100	48	321	✓
`jfs`	2	100	48	318	✓
`btrfs`	2	104	50	324	✓
`xfs`	-	-	-	-	-

(b) Dentry cache

FS	**Path**		**Bucket**	**Time (s)**	**Collide?**
	#	# OP	# OP		
All 6 FSes	21	806	117	604	✓

backward slicing—SIDEFINDER can tell that `xfs` should never be vulnerable from inode cache as it never uses them.

In the case of dentry cache, SIDEFINDER were able to identify the source location, which is the `filename` parameter of `open()` syscall. As we will elaborate more later, we found that the depth between the source and sink were not long, so SIDEFINDER does not suffer from aliasing issues for the dentry cache.

Note, the results from this program slicing phase do not provide any detailed semantics on how the input values will be used in computing the bucket index in a hash function.

Synthesizing attack inputs. In order to understand the effectiveness of SIDEFINDER in synthesizing sufficient number of collision inputs, we run SIDEFINDER's concolic execution until it finds 2,000 collision inputs. More specifically, we first prepared file system images for all six file systems, where each file system is populated with five files. Then per each file system, we performed the following steps: (1) we boot up the Linux kernel within the concolic execution engine of SIDEFINDER; (2) mount the file system; (3) instrument the true taint source point of the corresponding file system so that the respective source data is symbolized; (4) invoke `open()` syscall with the filename that we populated before, which in turn will activate the symbolization from the true taint source point; and (5) once the concolic execution reaches the sink point (i.e., a bucket index computation point), we ask a solver to find 2,000 unique values that are evaluated to the same bucket index.

19 shows the results on synthesizing on inode and dentry cache. Most importantly, SIDEFINDER were able to synthesize 2,000 colliding inputs within a reasonable amount of time, six minutes for inode cache and eleven minutes for dentry cache. As noted before, `xfs` is an exception here because `xfs` never uses inode cache. We also confirmed that these synthesized colliding inputs, `inode` in the case of inode cache and `filename` in the case of dentry cache, are actually colliding by dynamically running the kernel. In the next subsection, we further show our attack evaluations based on these synthesized inputs.

Moreover, we also want to emphasize that manually reverse-engineering this logic would be quite complicated as represented in the number of path and bucket constraints as well as the number of total operations in constraints. Especially for the dentry case, which involves complex string hash computations onto the filename to determine the bucket index, the total number of operations are very large (about eight times in path constraints and two times in a bucket constraint) compared to those for the inode cache. Thus, we believe SIDEFINDER's semi-automation would help developers to confirm the timing side-channel with its synthesized inputs.

To evaluate the effectiveness of SIDEFINDER in terms of speed, we also developed a simple brute-forcing technique finding collision inputs for the inode cache. In general, a brute-forcing attack is performed when an attacker wishes to minimize the manual analysis (e.g., instead of developing an inverse of the hash function) and launch the attack without complete understandings on the target. Following this common practice, our brute-forcing attack against the inode cache works as follows: (1) the attacker directly modifies `inode_hash()` in the Linux Kernel to log which bucket index is mapped for which file once

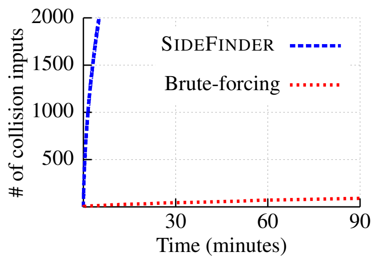
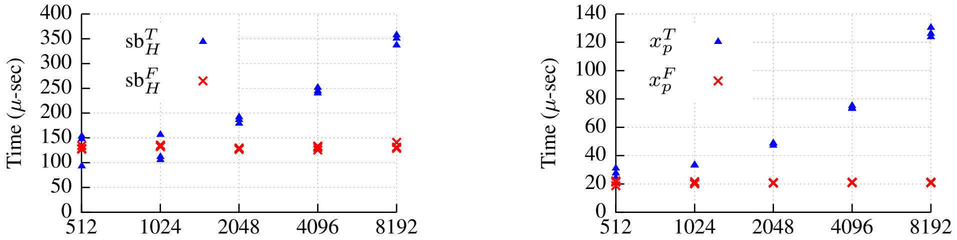


Figure 23: Comparing effectiveness of finding collision inputs on the inode cache using SIDEFINDER and brute-forcing techniques. The size of the inode cache was 2^{14}. We only show the ext2 case here as all other cases showed the similar results. SIDEFINDER found 2,048 collision inputs within 6 minutes while the bruteforcing found 129 collisions in 90 minutes.



(a) Inode cache

(b) Dentry cache

Figure 24: Attack results on two hash tables: (a) Inode cache. `sb`$_H$ on the superblock's address (`sb`), we inserted k files (X-axis) having the colliding inode numbers on `sb`$_H$. Then measured the averaged execution time to insert those k files (Y-axis). If `sb`$_H$ $=$ `sb`, the time take to insert k files keeps increasing as it needs to traverse a long chain of linked lists. If it is not, then the searching time is largely constant on k, which is the general property of a hash table — O(1) search time on the average case; (b) Dentry cache. Relying on a hypothetical filename x_p for the private file, we first opened k files (X-axis) having the colliding string hash value on x_p. Then we opened the same set of k files again, and measured the averaged execution time for each (Y-axis). If x_p is a correct, then the execution time keeps increasing over k because it needs to walk through a long chain of linked lists. However, if it is not, the execution time stays the same as there is no bucket collisions.

the function is invoked; (2) the attacker randomly creates a file, where a backing filesystem is mounted as `ext4`, using a user-space program to trigger `inode_hash()`; (3) the attacker monitors the log from the Kernel, and go back to step (1) until the sufficient number of collisions found.

As shown in Figure 23, SIDEFINDER performs significantly better than brute-forcing—while SIDEFINDER found all 2,000 colliding inputs within 6 minutes, the brute-forcing only found 129 in 90 minutes.

4.5.2 Attack Evaluation

Experimental settings. The evaluation for inode cache and dentry cache was performed on Linux Kernel 4.6-rc7 (x86-64) in KVM. To prepare a file system image to attack inode cache, we created files, each of which corresponds to the colliding inode number based on a set of colliding inode numbers generated by SIDEFINDER, A simple, but slow, way would be to create empty files until disk becomes full and then delete files with non-colliding inode numbers by checking their inode number using a `stat()` system call. As we found this process cannot scale (due to the limit on inode numbers) and is too slow, we implemented to create such files by directly modifying file system metadata relying on `debugfs` [42] on an ext4 disk image [106]. Note, to mount such a prepared disk image, there are several well-known approaches to mount a disk image with a non-privileged user (e.g., using USB memory stick). We triggered this attack by running our `ls`-like program, which iterates and opens a file on the disk image and reports its running time. We measured its running time using `clock()` system call. For dentry cache attacks, we used similar techniques as inode cache attacks. The difference is in that it does not need to create an actual file, as we use a negative dentry to insert a dentry object.

Inode cache. Recall that inode cache uses a combination of superblock address and inode number as the hash function input, and the attacker's goal is to infer the superblock address `sb`. To show the complete timing channel attacks (i.e., fully infer the superblock's address), we conduct and evaluate the attacks in the following three steps: (1) we launch the timing channel attacks on two primitive hypotheses (i.e., the correct and incorrect hypotheses) to determine the collision chain length exhibiting acceptable execution time differences; (2) we evaluate how the searching space on superblock addresses can be reduced due to constraints on the running system's configuration or environment; (3) given the collision chain length from (1) and the reduced searching space from (2), we setup a series of hypotheses and test these to fully infer the superblock address.

First, in order to see how execution time changes on correct or incorrect hypotheses, we measure the execution time with the following two primitive cases under the assumption that the true value the superblock is $\texttt{sb} = 0\text{xffff8800bb890930}$: the correct hypothesis value is 0xffff8800bb890930, which we refer as $\texttt{sb}_H^T$; and the incorrect hypothesis value is 0xffffffffbbbb0000, which we refer as $\texttt{sb}_H^F$. Figure 24(a) shows the averaged execution time (Y-axis, CPU clocks) to open the file after inserting k files (X-axis) on each hypothesis value, where the size of inode cache is 2^{13}. In case of $\texttt{sb}_H^T$, all k files will fall into the same bucket and cause their corresponding inode objects being connected together in a linked list. Thus, the execution time (Y-axis) increases as k increases (X-axis). In case of $\texttt{sb}_H^F$, k files are falling into different buckets as their hash values would be different with a high probability. Thus, the execution time stays relatively the same for different k values, indicating the general property of a hash table holds here — O(1) search time on the average case. Furthermore, as shown in Figure 24(a), when 2,048 files are colliding altogether, there is sufficient execution time differences to capture (i.e., on average, the execution time takes 45% more on the true hypothesis compared to the false hypothesis). Thus, in the following attack steps, we choose to insert 2,048 numbers to test each hypothesis.

Second, to reduce the searching space on the superblock address, we developed the strategy similar to the attacker's typical bruteforcing techniques commonly used in breaking ASLR. This strategy is largely relying on the constraints on the superblock address under the running system's configuration or environment. Overall, the superblock for the root partition is usually allocated in an early stage of kernel initialization. Combining this with the fact that the allocator (`slab` and the underlying page allocator) for the superblock is deterministic, we found that the search space can be reduced to a very small range of addresses. we assume that the attacker can obtain the basic system information including the kernel version, RAM sizes, etc, each of which can affect the kernel address layout and randomness. Thus, the attacker can setup the identical system configurations and environments in terms of the kernel address layouts, and further examine low level system information of the kernel. Moreover, this assumption is general and mostly true in many running Linux systems, because we already assumed that the attacker can run user-level code on a target machine in §4.2.1 and such information is mostly accessible from user-level code. In particular, to see how this strategy would help the attacker under our experimental setting, we rebooted the kernel 200 times and collect 200 true sb addresses. All of these values were always located in one of two different ranges, (S1, E1) = (0xffff8800bb808930, 0xffff8800bb8e8930) and (S2, E2) = (0xffff88042c9e8930, 0xffff88042cf98930), and always ended with 0x930. Thus, with a high probability, a possible sb value would be values obtained by iterating over above two ranges, which results in 1,680 distinct addresses.

Note, from this second step, we assume that the attacker can obtain the basic system information including the kernel version, RAM sizes, etc, each of which can affect the kernel address layout and randomness. Thus, the attacker can setup the identical system configurations and environments in terms of the kernel address layouts, and further examine low level system information of the kernel without any debugging capabilities on the target machine. This assumption is general and mostly true in many Linux running systems, because we already assumed that the attacker can run user-level code on a target machine in §4.2.1 and such information is accessible from user-level code.

Lastly, we finally carry out the full scan over the superblock address based on the following two information from the previous steps: (1) the collision chain length (i.e., inserting 2,048 files introduces sufficient execution time differences if the hypothesis is correct); (2) the reduced searching space on the superblock address (i.e., 1,680 distinct addresses are a potential address of the superblock). Given this information, we setup 1,680 hypotheses on each potential address of the superblock, and then insert 2,048 files to test each hypothesis. Among the tests on these 1,680 hypotheses, there were noticeable execution time differences when $\texttt{sb}_H$ is 0xffff8800bb890930, meaning that the attacker can correctly infer the full address information of the superblock. Moreover, completely testing all 1,680 hypotheses took 78.4 minutes (i.e., testing each hypothesis took 2.8 seconds on average).

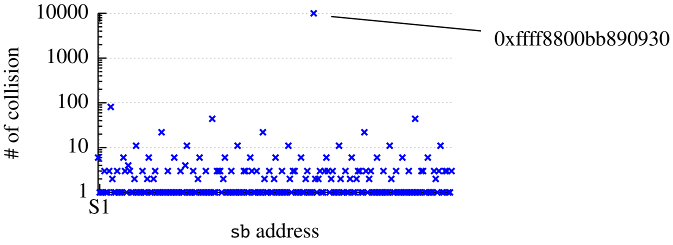


Figure 25: The number of colliding inode numbers in the expected bucket (Y-axis) for all possible hypothesis values of sb (X-axis). $\texttt{sb}_H$ is the only address value having significantly more collisions (i.e., 10,000) than other hypotheses (the second biggest less than 1,000). Since the execution time differences between 10,000 and 1,000 is easy observable, once attackers observe an execution time difference under $\texttt{sb}_H$, they can be certain on the address value of sb.

As we have shortly described in §4.4.1, it is possible that an incorrect hypothesis value of sb may also cause bucket collisions (i.e., the execution time may increase over k even if $\texttt{sb}_H \neq \texttt{sb}$). However, in practice, we found that this unexpected event would not occur largely due to the reduced searching space from step (2). In particular, For each of 1,680 hypotheses, we inserted 10,000 potential colliding inode numbers and counted the maximum number of inode that did end up in the expected bucket. The result is shown in Figure 25. Although the maximum number of collision can also be large for some incorrect hypotheses (e.g., 1,000), the number of the correct hypothesis $\texttt{sb}_H$ is significantly higher (10,000). Because traversing 10,000 inode objects in a linked list is much slower than traversing 1,000 inode objects (more than two times, see §4.4.1, we conclude that attackers can be certain on sb value once they capture the execution time differences.

Dentry cache in the Linux Kernel. In the dentry cache attack, the attacker's goal is to infer the secret filename based on the hypothetical filename x_p. To check if how much execution time changes on correct or incorrect hypotheses, we set up two hypothetical filenames, x_p^T is 'testmetestme' and x_p^F is 'debugmedebugme', when x_p is 'testmetestme'. Figure 24(b) shows the averaged execution time (Y-axis, CPU clocks) to open the file after inserting k files (X-axis) on each case, where the size of dentry cache is 2^{14}. In case of x_p^T, all k dentry objects will be placed into the same bucket, and they are stored as a linked list, and thus the value on Y-axis increases over the value on X-axis. However, in case of x_p^F, k files will be falling into different buckets with a high probability, and thus the execution time

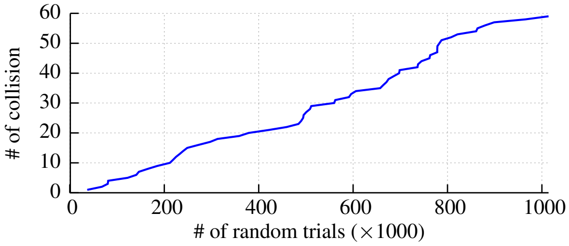


Figure 26: To see the accuracy on dentry cache attacks (see §4.4.2), we checked a random string if it can be accidentally fall into the same bucket index as the hypothetical file name x_p^T.

is constant over X-axis. Note that we have not prepared and created any file for dentry cache attack, because a negative entry will be inserted even if we try to open non-existing file.

Similar to dentry cache attacks, it is possible that some other private filename (i.e., $x_p^T \neq x_p$) may cause the bucket collisions. To see how frequent this accidental event would happen, we first created a random string r such that $r \neq x_p^T$ and the length of a string is within [5,20]. Then we check whether the bucket index of r would be the same as x_p^T (i.e., `d_hash(parent,` r`)` = `d_hash(parent,` x_p^T`)` assuming their parent directory is the same). Figure 26 illustrates the cumulative frequencies on the number of random strings checked (X-axis) and the number of accidental matches (Y-axis). When the number of random strings are 1 million, then their were about 60 accidental matches. This implies that the decision concluded on x_p^T relying on execution time slowdowns would have about 99.99% accuracy on the randomly distributed string. Although this accuracy rate would change relying on the distribution of private filename that the attacker wishes to infer, but we believe it would still give high accuracy as our empirically approximated accuracy on random distribution is very high.

4.6 Discussion on Mitigation Techniques

In this section, we discuss several possible mitigation techniques to timing-channel attacks. It is worth noting that the adoption of these mitigation approaches needs to consider specific runtime requirements as hash tables are usually used for performance-critical code.

No sensitive data for bucket index computation. The most obvious mitigation would be to never use sensitive data to compute a bucket index. On the other hand, there are very legitimate reasons, in particular, performance, to use sensitive data in the first place. For example, a design decision behind using `sb` and `parent` for inode and dentry caches, respectively, is to minimize hash table collisions for performance.

Heterogeneous hash tables. The root cause of this vulnerability that the same hash table is used for both user-controllable data (e.g., inode numbers or filenames) and sensitive data (e.g. an address value or privileged user's filename). If each hash table is dedicated for each type of data, there would be no execution dependencies at all between them and thus there will be no side-channels. However, as hash tables may require some dependent operations [108], simply separating tables possibly break functionality. [108] further proposes unique representation [109] based hash tables, formally verified leak-free heterogeneous hash tables.

Universal hash functions. The precondition of timing-channel attacks is in that an adversary should be able to obtain a set of inputs colliding with each other. Therefore, once this mapping is unknown to an adversary, timing-channel attacks can be mitigated. For example, in universal hash functions [14, 22], a hash function is randomly selected from a family of hash functions, at the beginning of each run.

Non-deterministic data structures. Replacing deterministic data structures with non-deterministic ones gives probabilistic guarantee of security. Thus, if additional performance costs are not the primary concern, non-deterministic data structures [13, 100] can be alternatives to deterministic ones.

4.7 Related work

Side-channel attacks and defenses. Previous work has discovered timing channels in implementations of cryptographic functions that can be exploited to infer sensitive information, such as a cryptographic key [16, 69]. Such attacks typically require significant manual effort to derive. In contrast, we have presented attacks that target use of hash tables in general applications. We have identified a class of timing channels that is sufficiently flexible to describe timing channels in widely-used system code, but simple enough that the program analysis required to find instances of such channels can be significantly automated.

Previous work has developed formal models and languages that can express notions of information flow in the presence of an attacker who can observe the execution time of a program [49, 82, 127]. Such approaches, while powerful, have not yet been applied to analyze practical systems code; however, we believe that such approaches could be adapted to enable programmers to check for and eliminate the class of timing-channel vulnerabilities that we have identified.

Previous work has presented side channels and techniques to mitigate them for specific execution platforms, in particular hardware [35, 70], web applications [9, 23, 128], and the cloud [67]. Previous work has also presented side channels that leak information over channels distinct from execution time, in particular system caches [44, 67, 70], features of network traffic [9, 23, 128], and program memory [60]. Compared to these work, we have presented timing channels for a distinct and ubiquitous execution platforms: arbitrary programs that use deterministic hash tables.

Focusing on a hash table, previous work has presented to intentionally trigger the worst case of algorithmic complexities in a hash table, which resulted in Denial-of-Service attacks in various applications [7, 37, 68]. Moreover, [93] first demonstrated the timing side-channel attacks against a hash table in Firefox, which also relies on the deterministic behavior of hash tables. Inspired by these findings, this thesis explores general property of the timing side-channel attacks against hash tables and further presents more complete and various case studies with new timing channel vulnerabilities.

Information Disclosure Attacks Previous work has presented techniques that bypasses address space layout randomization (ASLR) by derandomizing the addresses of critical memory objects. In particular, previous work has shown that particular classes of memory errors can be exploited to derandomize systems that do not use a sufficient number of bits for entropy [102], or keep traversing to extend the address layout knowledge [104]. As yet another direction, performance oriented designs may allow an attacker to bypass ASLR [72].

Program slicing *Program-slicing* problems are concerned with determining which program objects may influence each other during the execution of a program [57, 123]. A *backwards slice* of `x` is a complete set of program variables and instructions that may affect the value stored in `x`. The accuracy of program slices is determined by the underlying dependency analysis that they use: dependency analyses can be designed with varying flow, context, and field sensitivities [57, 123]; higher sensitivities trade performance for accuracy of the results that they produce. We have designed a backwards slicer that uses a dependency analysis that is flow-insensitive and context-sensitive, in response to our study of systems code.

Symbolic execution. Symbolic-execution engines have been applied, particularly in the context of software security, to find subtle bugs that depend heavily on the semantics of instructions and checks in the program [8, 17, 19, 20, 26, 27, 50, 98]. Those studies on symbolic execution are applied to check if a program satisfies a *safety* property based on symbolic summaries of functions; i.e., the engines check if there is *some* state of the program that, over a *single* run, drives the program to an insecure state (although the engines can potentially generate multiple states that each violate a given safety property). Our approach also uses a symbolic execution engine, but instead it solves HTCA problem to synthesize an attack on a hash table client.

CHAPTER V

CONCLUSION AND FUTURE WORK

In this thesis, we developed tools to protect computer systems through eliminating or analyzing vulnerabilities. In order to eliminate use-after-free vulnerabilities, We developed DANGNULL, applied it to Chromium, and conducted a thorough evaluation showing the effectiveness and compatibility of DANGNULL. We also developed CAVER, a runtime bad-casting detection tool which verifies type casting dynamically. Lastly, we developed SIDEFINDER, which can synthesize concrete inputs attacking timing-channel vulnerabilities in hash tables, that helps testing and confirming processes. In order to demonstrate its broader and practical impacts, we applied these techniques to popular commodity software such as web browsers and the Linux kernel. We also evaluated these based on real-world attacks and showed its security effectiveness.

In the future, the further optimization techniques of DANGNULL and CAVER can we a potential research direction. Since much runtime overheads of these tools are caused by metadata accesses in the data structure (i.e., a red-black tree), we may try other types of data structures. One potential design is to append the metadata before or after an object. Note, while using this design would bring some runtime speed benefits as the complexity of an access will be constant (i.e., O(1)), it may impose heavy memory uses because of strictly aligned virtual memory layouts for objects.

Because our methods in DANGNULL and CAVER are per vulnerability elimination, this thesis does not cover memory corruption vulnerability classes other than use-after-free and bad-casting. Thus, other popular types of vulnerabilities including, heap overflow or uninitialized memory read/write, are not covered. We believe these types of vulnerabilities can be also eliminated in the future. For example, in the case of heap overflows, the concept of a boundless heap (i.e., an object virtually allocated with an infinite size) has a potential to be used for heap-overflow eliminations. However, since this needs to implement additional virtual layers on the memory access, it has to be carefully carried out to be practical.

REFERENCES

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., and LIGATTI, J., "Control-flow integrity," in *CCS*, 2005.

[2] ABHISHEK ARYA, C. N., "Fuzzing for Security (The Chromium Blog)." `http://blog.chromium.org/2012/04/fuzzing-for-security.html`, 2012.

[3] AKRITIDIS, P., "Cling: A Memory Allocator to Mitigate Dangling Pointers," in *USENIX Security Symposium (Security)*, 2010.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., and HAND, S., "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors," in *USENIX Security Symposium (Security)*, 2009.

[5] ALEXA, "The Top 500 Sites on the Web." `http://www.alexa.com/topsites`, Aug 2014.

[6] ASHCRAFT, K. and ENGLER, D. R., "Using programmer-written compiler extensions to catch security holes," in *IEEE SP*, 2002.

[7] AUMASSON, J.-P., BERNSTEIN, D. J., and BOBLET, M., "Hash-flooding DoS reloaded: attacks and defenses," in *29th Chaos Communications Congress*, 2012.

[8] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., and BRUMLEY, D., "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[9] BACKES, M., DOYCHEV, G., and KÖPF, B., "Preventing Side-Channel Leaks in Web Traffic: A Formal Approach," in *NDSS*, 2013.

[10] BAD CASTING: FROM BASICTHEBESLAYER TO BASICCONTAINERLAYER, "Mozilla Bugzilla - Bug 1074280." `https://bugzilla.mozilla.org/show_bug.cgi?id=1074280`, Nov 2014.

[11] BAD-CASTING FROM PRCLISTSTR TO NSSHISTORY, "Mozilla Bugzilla - Bug 1089438." `https://bugzilla.mozilla.org/show_bug.cgi?id=1089438`, Nov 2014.

[12] BERGER, E. D. and ZORN, B. G., "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[13] BETHEA, D. and REITER, M. K., "Data structures with unpredictable timing," in *ESORICS*, 2009.

[14] BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., and ROGAWAY, P., "UMAC: Fast and secure message authentication," in *CRYPTO*, 1999.

[15] BLINK : THE RENDERING ENGINE USED BY CHROMIUM. http://www.chromium.org/blink, Aug 2014.

[16] BRUMLEY, D. and BONEH, D., “Remote timing attacks are practical,” in *USENIX Security Symposium*, 2003.

[17] BRUMLEY, D., POOSANKAM, P., SONG, D. X., and ZHENG, J., “Automatic patch-based exploit generation is possible: Techniques and implications,” in *IEEE SP*, 2008.

[18] CABALLERO, J., GRIECO, G., MARRON, M., and NAPPA, A., “Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[19] CADAR, C., DUNBAR, D., and ENGLER, D. R., “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, 2008.

[20] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., and ENGLER, D. R., “EXE: automatically generating inputs of death,” in *CCS*, 2006.

[21] CARLINI, N. and WAGNER, D., “ROP is still dangerous: Breaking modern defenses,” in *USENIX Security Symposium (Security)*, 2014.

[22] CARTER, J. L. and WEGMAN, M. N., “Universal Classes of Hash Functions (Extended Abstract),” in *STOC*, 1977.

[23] CHAPMAN, P. and EVANS, D., “Automated black-box detection of side-channel vulnerabilities in web applications,” in *CCS*, 2011.

[24] CHEN, H. and WAGNER, D., “MOPS: an infrastructure for examining security properties of software,” in *CCS*, 2002.

[25] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., and IYER, R. K., “Non-control-data Attacks Are Realistic Threats,” in *USENIX Security Symposium (Security)*, 2005.

[26] CHIPOUNOV, V., KUZNETSOV, V., and CANDEA, G., “S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems,” in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[27] CHO, C. Y., BABIC, D., POOSANKAM, P., CHEN, K. Z., WU, E. X., and SONG, D., “MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery,” in *USENIX Security Symposium*, 2011.

[28] CHRIS EVANS, “Using ASAN as a protection.” http://scarybeastsecurity.blogspot.com/2014/09/using-asan-as-protection.html, Nov 2014.

[29] CHROMIUM, “Chromium project: Log of /trunk/src/tools/ubsan_vptr/blacklist.txt.” https://src.chromium.org/viewvc/chrome/trunk/src/tools/ubsan_vptr/blacklist.txt?view=log, Nov 2014.

[30] CHROMIUM, "Chromium Revision 285353 - Blacklist for UBsan's vptr." https://src.chromium.org/viewvc/chrome?view=revision&revision=285353, Jul 2014.

[31] CHROMIUM PROJECTS, "Running Tests at Home." http://www.chromium.org/developers/testing/running-tests, Aug 2014.

[32] CLANG DOCUMENTATION, "MSVC Compatibility." http://clang.llvm.org/docs/MSVCCompatibility.html, Nov 2014.

[33] CODESOURCERY, COMPAQ, EDG, HP, IBM, INTEL, HAT, R., and SGI, "Itanium C++ ABI (Revision: 1.83)." http://mentorembedded.github.io/cxx-abi/abi.html, 2005.

[34] CODESOURCERY, COMPAQ, EDG, HP, IBM, INTEL, RED HAT, AND SGI, "C++ ABI Closed Issues." www.codesourcery.com/public/cxx-abi/cxx-closed.html, Nov 2014.

[35] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., and SUTTER, B. D., "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *IEEE SP*, 2009.

[36] CRISWELL, J., LENHARTH, A., DHURJATI, D., and ADVE, V., "Secure virtual architecture: A safe execution environment for commodity operating systems," in *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[37] CROSBY, S. A. and WALLACH, D. S., "Denial of service via algorithmic complexity attacks.," in *Usenix Security*, 2003.

[38] DAI MIKURUBE, "C++ Object Type Identifier for Heap Profiling." http://www.chromium.org/developers/deep-memory-profiler/cpp-object-type-identifier, Nov 2014.

[39] DANIEL, M., HONOROFF, J., and MILLER, C., "Engineering Heap Overflow Exploits with JavaScript," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[40] DAVI, L., LEHMANN, D., SADEGHI, A.-R., and MONROSE, F., "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *USENIX Security Symposium (Security)*, 2014.

[41] DE MOURA, L. M. and BJØRNER, N., "Z3: an efficient SMT solver," in *TACAS*, 2008.

[42] "debugfs(8) - Linux man page," 2015. http://linux.die.net/man/8/debugfs.

[43] DHURJATI, D. and ADVE, V., "Efficiently Detecting All Dangling Pointer Uses in Production Servers," in *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[44] DOYCHEV, G., FELD, D., KÖPF, B., MAUBORGNE, L., and REINEKE, J., "Cacheaudit: A tool for the static analysis of cache side channels," in *USENIX Security Symposium*, 2013.

[45] FEIST, J., MOUNIER, L., and POTET, M.-L., "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, 2013.

[46] FLAK, "Analysis of OpenSSL Freelist Reuse." `http://www.tedunangst.com/flak/post/analysis-of-openssl-freelist-reuse`, Aug 2014.

[47] GANESH, V. and DILL, D. L., "A decision procedure for bit-vectors and arrays," in *CAV*, 2007.

[48] GCC, "Arrays of Variable Length." `https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html`, Jun 2015.

[49] GIACOBAZZI, R. and MASTROENI, I., "Timed Abstract Non-Interference," in *FORMATS*, 2005.

[50] GODEFROID, P., KLARLUND, N., and SEN, K., "DART: directed automated random testing," in *PLDI*, 2005.

[51] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., and PORTOKALIDIS, G., "Out of control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[52] GÖKTAŞ, E., ATHANASOPOULOS, E., POLYCHRONAKIS, M., BOS, H., and PORTOKALIDIS, G., "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *USENIX Security Symposium (Security)*, 2014.

[53] GOOGLE, "Octane Benchmark." `https://code.google.com/p/octane-benchmark`, Aug 2014.

[54] GOOGLE, "Specialized memory allocator for ThreadSanitizer, MemorySanitizer, etc.." `http://llvm.org/klaus/compiler-rt/blob/7385f8b8b8723064910cf9737dc929e90aeac548/lib/sanitizer_common/sanitizer_allocator.h`, Nov 2014.

[55] HASTINGS, R. and JOYCE, B., "Purify: Fast detection of memory leaks and access errors," in *Winter 1992 USENIX Conference*, 1991.

[56] HERLIHY, M. and MOSS, J. E. B., *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[57] HORWITZ, S., REPS, T. W., and BINKLEY, D., "Interprocedural slicing using dependence graphs," in *PLDI*, 1988.

[58] HP, "Pwn2Own 2014: A recap." `http://www.pwn2own.com/2014/03/pwn2own-2014-recap`, Aug 2014.

[59] HUND, R., WILLEMS, C., and HOLZ, T., "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *IEEE SP*, 2013.

[60] JANA, S. and SHMATIKOV, V., "Memento: Learning secrets from process footprints," in *IEEE SP*, 2012.

[61] JANG, D., TATLOCK, Z., and LERNER, S., "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[62] JANG, D., TATLOCK, Z., and LERNER, S., "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[63] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., and PORTOKALIDIS, G., "ShadowReplica: efficient parallelization of dynamic data flow tracking," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[64] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., and WANG, Y., "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference (ATC)*, 2002.

[65] JOSHI, S., LAHIRI, S. K., and LAL, A., "Underspecified harnesses and interleaved bugs," in *POPL*, 2012.

[66] JTC1/SC22/WG21, I., "ISO/IEC 14882:2013 Programming Language C++ (N3690)." `https://isocpp.org/files/papers/N3690.pdf`, 2013.

[67] KIM, T., PEINADO, M., and MAINAR-RUIZ, G., "StealthMem: System-level protection against cache-based side channel attacks in the cloud," in *USENIX Security Symposium*, 2012.

[68] KLINK, ALEXANDER AND WÃĎLDE, JULIAN, "Efficient Denial of Service Attacks on Web Application Platforms," in *28th Chaos Communication Congress*, 2011.

[69] KOCHER, P. C., "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*, 1996.

[70] KÖPF, B. and BASIN, D., "An Information-Theoretic Model for Adaptive Side-Channel Attacks," in *CCS*, 2007.

[71] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., and SONG, D., "Code-pointer integrity," in *OSDI*, 2014.

[72] LEE, B., JANG, Y., WANG, T., SONG, C., LU, L., KIM, T., and LEE, W., "Abusing performance optimization weaknesses to bypass ASLR," in *Black Hat*, 2014.

[73] LEE, B., LU, L., WANG, T., KIM, T., and LEE, W., "From Zygote to Morula: Fortifying weakened ASLR on android," in *IEEE SP*, 2014.

[74] LEISERSON, C. E., RIVEST, R. L., STEIN, C., and CORMEN, T. H., *Introduction to algorithms*. MIT press, 2009.

[75] LLVM PROJECT, "LLVM Language Reference Manual." http://llvm.org/docs/LangRef.html.

[76] LLVM PROJECT, "How to set up LLVM-style RTTI for your class hierarchy." http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html, Nov 2014.

[77] LLVMLINUX, "LLVMLInux Project." http://llvm.linuxfoundation.org/index.php/Main_Page.

[78] LONG, F., SIDIROGLOU-DOUSKOS, S., and RINARD, M., "Automatic Runtime Error Repair and Containment via Recovery Shepherding," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[79] MADS AGER, ERIK CORRY, VYACHSLAV EGOROV, KENTARO HARA, GUSTAV WIBLING, IAN ZERNY, "Oilpan: Tracing Garbage Collection for Blink." http://www.chromium.org/blink/blink-gc, Aug 2014.

[80] MALLOCAT, "Subverting without EIP." http://mallocat.com/subverting-without-eip, Aug 2014.

[81] MIHALICZA, J., PORKOLÁB, Z., and GABOR, A., "Type-preserving heap profiler for c++," in *IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[82] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., and WAGNER, D., "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2006.

[83] MOZILLA, "DROMAEO, JavaScript Performance Testing." http://dromaeo.com, Aug 2014.

[84] MULTIPLE UNDEFINED BEHAVIORS (STATIC_CAST<>) IN LIBSTDC++-V3/INCLUDE/BITS, "GCC Bugzilla - Bug 63345." https://gcc.gnu.org/bugzilla/show_bug.cgi?id=63345, Nov 2014.

[85] MWR LABS, "MWR Labs Pwn2Own 2013 Write-up: Webkit Exploit." https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/, 2013.

[86] NAGARAJU, S., CRAIOVEANU, C., FLORIO, E., and MILLER, M., "Software Vulnerability Exploitation Trends," *Microsoft*, 2013.

[87] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., and ZDANCEWIC, S., "CETS: Compiler Enforced Temporal Safety for C," in *International Symposium on Memory Management (ISMM)*, 2010.

[88] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., and ZDANCEWIC, S., “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[89] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., and WEIMER, W., “Ccured: Type-safe retrofitting of legacy software,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[90] NECULA, G. C., MCPEAK, S., and WEIMER, W., “Ccured: type-safe retrofitting of legacy code,” in *POPL*, 2002.

[91] NETHERCOTE, N. and SEWARD, J., “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 89–100, 2007.

[92] NOVARK, G. and BERGER, E. D., “DieHarder: Securing the Heap,” in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[93] PAKT, “Leaking information with timing attacks on hashtables, part 1,” Aug. 2012. https://gdtr.wordpress.com/2012/08/07/leaking-information-with-timing-attacks-on-hashtables-part-1.

[94] POST, H. and KÜCHLIN, W., “Integrated static analysis for linux device driver verification,” in *Integrated Formal Methods*, pp. 518–537, Springer, 2007.

[95] PRANDINI, M. and RAMILLI, M., “Return-oriented programming,” *IEEE Security & Privacy*, 2012.

[96] QUALYS SECURITY ADVISORY, “Qualys Security Advisory CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow.” http://www.openwall.com/lists/oss-security/2015/01/27/9, Jun 2015.

[97] RATANAWORABHAN, P., LIVSHITS, B., and ZORN, B., “NOZZLE: A Defense Against Heap-spraying Code Injection Attacks,” in *USENIX Security Symposium (Security)*, 2009.

[98] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., and SONG, D., “A symbolic execution framework for javascript,” in *IEEE SP*, 2010.

[99] SEACORD, R., *Secure Coding in C and C++*. Addison-Wesley Professional, first ed., 2005.

[100] SEIDEL, R. and ARAGON, C., “Randomized search trees,” *Algorithmica*, vol. 16, no. 4-5, pp. 464–497, 1996.

[101] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., and VYUKOV, D., “AddressSanitizer: A Fast Address Sanity Checker,” in *USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[102] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E., MODADUGU, N., and BONEH, D., "On the effectiveness of address-space randomization," in *CCS*, 2004.

[103] SHENDE, S., MALONY, A. D., CUNY, J., BECKMAN, P., KARMESIN, S., and LINDLAN, K., "Portable profiling and tracing for parallel, scientific applications using c++," in *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1998.

[104] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., and SADEGHI, A.-R., "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE SP*, 2013.

[105] SOTIROV, A., "Heap Feng Shui in JavaScript," *Black Hat Europe*, 2007.

[106] "How do I create a file with a specific inode number?," 2011. `http://stackoverflow.com/questions/5752822/how-do-i-create-a-file-with-a-specific-inode-number`.

[107] STANDARD PERFORMANCE EVALUATION CORPORATION, "SPEC CPU 2006." `http://www.spec.org/cpu2006`, Aug 2014.

[108] STEWART, G., BANERJEE, A., and NANEVSKI, A., "Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures," in *PPDP*, 2013.

[109] SUNDAR, R. and TARJAN, R. E., "Unique binary search tree representations and equality-testing of sets and sequences," in *STOC*, 1990.

[110] THE CHROMIUM PROJECT. `http://www.chromium.org/Home`, Aug 2014.

[111] THE CHROMIUM PROJECT, "Chromium Issues - Bug 387016." `http://code.google.com/p/chromium/issues/detail?id=387016`, Nov 2014.

[112] THE CHROMIUM PROJECTS, "Chromium Issues." `https://code.google.com/p/chromium/issues`, Aug 2014.

[113] THE CHROMIUM PROJECTS, "Undefined Behavior Sanitizer for Chromium." `http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer`, Nov 2014.

[114] THE LLVM COMPILER INFRASTRUCTURE. `http://llvm.org`, Aug 2014.

[115] THE MOZILLA FOUNDATION, "Firefox Web Browser." `https://www.mozilla.org/firefox`, Nov 2014.

[116] THE WEB STANDARDS PROJECT, "Acid Tests." `http://www.acidtests.org/`, Aug 2014.

[117] THE WEBKIT OPEN SOURCE PROJECT. `http://www.webkit.org`, Aug 2014.

[118] TICE, C., “Improving Function Pointer Security for Virtual Method Dispatches,” in *GNU Tools Cauldron Workshop*, 2012.

[119] TIS COMMITTEE, “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2,” *TIS Committee*, 1995.

[120] UBUNTU, “0-address protection in Ubuntu.” `https://wiki.ubuntu.com/Security/Features#null-mmap`, Aug 2014.

[121] WALDE, J. and KLINK, A., “Effective Denial of Service attacks against web application platforms,” 2011. Chaos Communications Congress 28C3.

[122] WEBKIT, “SunSpider 1.0.2 JavaScript Benchmark.” `https://www.webkit.org/perf/sunspider/sunspider.html`, Aug 2014.

[123] WEISER, M., “Program slicing,” in *ICSE*, 1981.

[124] XU, W., DUVARNEY, D. C., and SEKAR, R., “An efficient and backwards-compatible transformation to ensure memory safety of C programs,” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[125] ZENG, B., TAN, G., and MORRISETT, G., “Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing,” in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[126] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., and ZOU, W., “Practical Control Flow Integrity and Randomization for Binary Executables,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[127] ZHANG, D., ASKAROV, A., and MYERS, A. C., “Language-based control and mitigation of timing channels,” in *PLDI*, 2012.

[128] ZHANG, K., LI, Z., WANG, R., WANG, X., and CHEN, S., “Sidebuster: automated detection and quantification of side-channel leaks in web application development,” in *CCS*, 2010.

[129] ZHANG, M. and SEKAR, R., “Control flow integrity for COTS binaries,” in *USENIX Security Symposium*, 2013.