Exploiting and Mitigating Advanced Security Vulnerabilities

by

Haehyun Cho

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved February 2021 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Co-Chair
Adam Doupé, Co-Chair
Yan Shoshitaishvili
Ruoyu Wang
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Cyberspace has become a field where the competitive arms race between defenders and adversaries play out. Adaptive, intelligent adversaries are crafting new responses to the advanced defenses even though the arms race has resulted in a gradual improvement of the security posture. This dissertation aims to assess the evolving threat landscape and enhance state-of-the-art defenses by exploiting and mitigating two different types of emerging security vulnerabilities.

I first design a new cache attack method named Prime+Count which features low noise and no shared memory needed. I use the method to construct fast data covert channels. Then, I propose a novel software-based approach, SmokeBomb, to prevent cache side-channel attacks for inclusive and non-inclusive caches based on the creation of a private space in the L1 cache. I demonstrate the effectiveness of SmokeBomb by applying it to two different ARM processors with different instruction set versions and cache models and carry out an in-depth evaluation.

Next, I introduce an automated approach that exploits a stack-based information leak vulnerability in operating system kernels to obtain sensitive data. Also, I propose a lightweight and widely applicable runtime defense, ViK, for preventing temporal memory safety violations which can lead attackers to have arbitrary code execution or privilege escalation together with information leak vulnerabilities. The security impact of temporal memory safety vulnerabilities is critical, but, they are difficult to identify because of the complexity of real-world software and the spatial separation of allocation and deallocation code. Therefore, I focus on preventing not the vulnerabilities themselves, but their exploitation. ViK can effectively protect operating system kernels and user-space programs from temporal memory safety violations, imposing low runtime and memory overhead.

ACKNOWLEDGMENTS

This dissertation could not have been made without the tremendous support.

I would like to express my deepest appreciation to my advisors and committee members: Prof. Ahn, Prof. Doupé, Prof. Shoshitaishvili, Prof. Wang, Prof. Bao, and Prof. Wu. All the help you gave me went above, which guided me to have a wealth of new knowledge and research insights. Also, your valuable advice throughout the whole process in my Ph.D. study have helped me to be an independent researcher.

My fellow SEFCOM members have helped me along the way. Your suggestions brought in threads of thought that made my research so much richer. You've also inspired me with your own body of research. I look forward to working with you again in the future, as our research interests will undoubtedly invite future collaboration.

I, also, would like give very special thanks to my Mom, Dad, and brother for their love and support.

And finally, my beloved wife Jooyeon, my Ph.D. journey simply could not have been accomplished without your help. Thanks for being not only an awesome friend, but also a special partner. Thanks again for being there for me.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Attackers and defenders have been fighting the "Eternal War in Memory" for a long time [132]. This cycle typically encompasses the creation (by defensive security researchers) of an effective mitigation again previously-known exploitation techniques, then, in response, the development (by offensive hackers) of new exploitation approaches to replace the mitigated ones. While this cat-and-mouse game may seem discouraging to researchers, it *has* resulted in a gradual improvement of the security posture of modern applications as stack-based buffer overflows were mitigated by non-executable stack memory and stack canaries, return-into-libc exploits were addressed by Address Space Layout Randomization (ASLR), and even return-oriented programming has been impacted by an unrelenting exploration of various defense techniques [127].

Despite such advances in system hardening have effectively defended software against many attacks based on memory corruption vulnerabilities, attackers are finding new attack surfaces, new types of vulnerabilities, and new offensive technologies to bypass state-of-the-art defense techniques. A survey on trends in the vulnerability landscape showed that the number of stack-based memory corruption vulnerabilities decreased dramatically from 2012, but the number of other types of vulnerabilities such as use-after-free, heap out-of-bounds read, and uses of uninitialized memory have generally increased [102]. In addition, the CPU caches are drawing attention as a new attack surface on non-Intel architectures. Cache-side channel attacks attacks are very effective in stealing cryptographic keys (*e.g.*, the secret keys used in AES) from victim programs and even other virtual machines, in tracing the execution of

programs, and in performing other malicious actions [91].

To assess the evolving threat landscape and enhance state-of-the-art defenses, this dissertation aims to exploit and mitigate two different types of emerging security vulnerabilities: cache side-channels and temporal memory safety violations. I first propose a new cache attack method to build data covert channels and a novel software-based approach to protect against cache side-channel attacks. This vulnerability class has broaden the cybersecurity boundaries and the bulk of research into cache attacks was started on, and has continued on, the Intel architecture. However, as mobile devices (*e.g.*, smartphones and watches) experience unprecedented growth, the mitigation of cache attacks on non-Intel architectures has drastically risen in importance. Thus, our community needs a defense against cache side-channel attacks that can protect against both shared cache (L2) attacks and dedicated cache (L1) attacks, and can do so regardless of the specific architecture in question. Next, I introduce a generic and automated exploitation approach for leaking sensitive information (*i.e.*, randomized memory addresses). Because modern systems deploy ASLR, it is unlikely to have arbitrary code execution or privilege escalation without learning randomized memory addresses. Lastly, I propose a practical runtime defense mechanism against temporal memory safety vulnerabilities in user-space applications and operating systems (OS) kernels. Currently, one of the most common vulnerabilities that can lead attackers to take over victim systems through arbitrary code execution or privilege escalation is a temporal memory safety vulnerability (*e.g.*, use-after-free). Therefore, preventing the security impact of temporal memory safety vulnerabilities is of great importance, yet it is still an open problem.

Building Data Covert Channels

The ARM TrustZone architecture, which provides hardware-assisted isolation, is widely adopted in mobile and IoT devices. The security of ARM TrustZone relies on the idea of splitting system-on-chip hardware and software into two worlds, namely normal world and secure world. Two legitimate channels exist at the hardware level that a normal world and a secure world component can use to communicate with each other.

In this work, we report cross-world covert channels, which exploit the world-shared cache in the TrustZone architecture. While it is easy to assume that cross-world covert channels must exist given that cache is shared, there is no comprehensive study on the practicality and bandwidth of cross-world covert channels in the TrustZone architecture. We design a Prime+Count technique that only cares about *how many* cache *sets* or *lines* have been occupied. The coarser-grained approach significantly reduces the noise introduced by the pseudo-random replacement policy and world switching. Using the Prime+Count technique, we build covert channels in single-core and cross-core scenarios in the TrustZone architecture. We test our implementations on two commercial devices (Samsung Tizen TV with ARMv7 CPU and Hikey board with ARMv8 CPU). The evaluation results show that the cross-world bandwidth could be as high as 27 KB/s in the single-core scenario using L1 cache and 95 B/s in the cross-core scenario using L2 cache. The results demonstrate that Prime+Count is an effective technique for enabling cross-world covert channels on ARM TrustZone.

Mitigating Cache Side-channel Attacks

Cache side-channel attacks abuse microarchitectural designs meant to optimize memory access to infer information about victim processes, threatening data privacy and security. Though many defenses have been proposed, no defense successfully protects against all proposed cache attacks without relying on architecture-specific features. Thus, there is no effective cache side-channel protection technique applicable to the increasingly growing share of non-Intel processors.

We propose SmokeBomb, a novel cache side-channel mitigation method that functions by explicitly ensuring a private space for each process to safely access sensitive data. The heart of the idea is to exclusively use the L1 cache of the CPU core as a *private space* by which SmokeBomb can give consistent results against cache attacks on the sensitive data. We demonstrate the effectiveness of SmokeBomb by applying it to two different ARM processors with different instruction set versions and cache models and carry out an in-depth evaluation of the efficiency of the protection. Therefore, attackers cannot distinguish specific data used by the victim. The experimental results show that SmokeBomb can prevent currently formalized cache attack methods effectively.

Leaking Sensitive Information of OS Kernels

Information leaks are the most prevalent type of vulnerabilities among all known vulnerabilities in Linux kernel. Many of them are caused by the use of uninitialized variables or data structures. It is generally believed that the majority of information leaks in Linux kernel are low-risk and do not have severe impact due to the difficulty (or even the impossibility) of exploitation. As a result, developers and security analysts do not pay enough attention to mitigating these vulnerabilities. Consequently,

these vulnerabilities are usually assigned low CVSS scores or without any CVEs assigned. Moreover, many patches that address uninitialized data use bugs in Linux kernel are not accepted, leaving billions of Linux systems vulnerable.

Nonetheless, information leak vulnerabilities in Linux kernel are not as low-risk as people believe. In this paper, we present a generic approach that converts stack-based information leaks in Linux kernel into kernel-pointer leaks, which can be used to defeat modern security defenses such as KASLR. Taking an exploit that triggers an information leak in Linux kernel, our approach automatically converts it into a highly impactful exploit that leaks pointers to either kernel functions or the kernel stack. We evaluate our approach on four known CVEs and one security patch in Linux kernel and demonstrate its effectiveness. Our findings provide solid evidence for Linux kernel developers and security analysts to treat information leaks in Linux kernel more seriously.

Assuring the Temporal Memory Safety

Temporal memory safety violations, such as use-after-free (UAF) vulnerabilities, are a critical security issue that can lead to exploitation of software that is developed in unsafe languages, such as C and C++. Unfortunately, unsafe languages still dominate the modern world of computing and are playing an integral role in ubiquitous software, such as OS kernels and user-space applications. Hence, defenses against temporal memory safety violations are of paramount importance.

In this work, we introduce ViK: a novel, lightweight, and widely applicable runtime defense that can protect both user-space applications and OS kernels against temporal memory safety violations. ViK automatically instruments programs at compile time via a compiler pass. The core idea of ViK is *object ID inspection*: ViK assigns a random identifier to every allocated object and stores the identifier in the *unused bits*

of the corresponding pointer. Before dereferencing each pointer, ViK inspects the pointer value to ensure that the pointer still references the original object for which it was created; Otherwise, a UAF violation is found. With careful optimizations that are performed during the static analysis phase, ViK overcomes the limitations of prior work and achieves low runtime overhead.

1.2 Contributions

In summary, this dissertation makes the following contributions.

- **Design of a new cache attack method.** We design a novel low noise, no shared memory needed cache attack named Prime+Count. Then, we use Prime+Count to construct fast data covert channels in the cross-world single-core and cross-core scenarios.

- **Novel defense against cache side-channel attacks.** We propose a novel software-based approach to mitigate cache side-channel attacks for inclusive and non-inclusive caches based on the creation of a private space in the L1 cache.

- **Exploitation of stack-based information leak vulnerability.** We identify common challenges in exploiting information-leak vulnerabilities. We, then, introduce a generic and automated approach that converts a stack-based information leak vulnerability in Linux kernel to an exploit that leaks kernel pointer values which can be directly used to attack victim systems with other types of vulnerabilities.

- **General approach for ensuring the temporal memory safety.** We propose ViK, a novel, lightweight, and widely applicable runtime defense to protect OS kernels and user-space programs against temporal memory safety violations. Its low runtime and memory overhead makes ViK applicable to real-world targets.

Chapter 2

BACKGROUND

In this chapter, I first present an overview of the ARM cache architecture. After presenting important concepts in the ARM cache architecture, I discuss some known cache attacks. In addition, I provide a technical overview of stack-based information leaks in OS kernels. Lastly, I introduce temporal memory safety violations.

2.1 ARM Cache Architecture

A cache is low-capacity low-latency memory located between a CPU and main memory. Because access to the cache is significantly faster than to main memory, the presence of caches dramatically improves the runtime performance of a system. Modern processors usually have two or more levels of hierarchical cache structure.

In the ARM architecture, each core has its own "L1" data *and* instruction cache. The separation of instruction and data cache allows transfers to be performed simultaneously on both instruction and data buses and increases the overall performance of L1 caches. In addition, all of the cores share a larger, unified "L2" cache. When the CPU needs data from a specific memory address, and that data is not in a cache (this is termed a *cache miss*), a cache linefill occurs. Otherwise, the CPU loads the data from its own L1 cache, its own L2 cache, or even the L1 cache of other CPUs using the directory protocol [78].

Set Associativity and Cache Addressing

Operations between a cache and main memory are done in chunks of a fixed size, called a *cache line*, for improved performance. A data memory address is divided

into three parts: tag, index, and offset. The index determines in which *cache set* the data should be stored. The tag contains the most significant bits of an address, which is stored in the cache along with the data so that the data can be identified by its address in main memory.

A *set-associative cache* is divided into several sets that consist of the same number of cache lines. The cache is called an *N-way associative cache* if a set has N cache lines, or *ways*. The data at a specific main memory address can be fetched into any cache line (*way*) of a particular set. In the ARM architecture, caches are always set-associative for efficiency reasons [29].

Either virtual or physical addresses can be used for the tag and index. In ARM CPUs, the L1 data cache is indexed using the physical address whereas the L1 instruction cache is indexed with the virtual address [27, 31]. The L2 cache is usually physically-indexed.

Replacement policy

In set-associative caches, to decide which specific cache line to use in a particular set several policies can be utilized: (1) *Least-recently-used replacement policy*: the least recently used cache entry in a cache set will be replaced. Intel architecture uses this policy [95]; (2) *Round-Robin replacement policy*: the cache lines that are first filled will be cleared first; (3) *Pseudo-random replacement policy*: a random cache line will be evicted. In the ARM architecture, a pseudo-random replacement policy is used, which usually makes cache-based attacks harder to implement [119, 91].

While the Intel architecture employs the least-recently-used (LRU) replacement policy [67], the ARM architecture generally uses a pseudo-random replacement policy [29]. However, some Cortex-A processors support other cache replacement policies, such as LRU policy and round-robin policy, which can be chosen by the system

developer [25, 40, 41, 35].

Inclusiveness

A cache architecture can be categorized based on whether or not a higher-level cache continues to hold data loaded into a lower-level cache. In *inclusive* caches, cache lines of the L2 cache will not be evicted by new data as long as the data is stored in the L1 cache, which is called *AutoLock* in prior work [66]. In *non-inclusive* caches, a line in L2 cache can be evicted to make space for new data even if the line is present in L1 cache. In *exclusive* caches, there is only one copy of data in the whole cache hierarchy—that is, when a line is loaded into L1, it is flushed from L2. Most ARM processors employ a non-inclusive cache [91].

By reviewing the technical reference manuals of the Cortex-A series [32, 26, 21, 24, 25, 28, 40, 41, 33, 42, 34, 35, 36, 43], I confirmed the inclusiveness of the following Cortex-A CPUs: (1) only A55 is exclusive;[1] (2) A15, A57, and A72 are inclusive;[2] and (3) A53 is non-inclusive.[3] Judging from the manuals, the other CPUs also would use the non-inclusive cache as the Cortex-A53.

2.2 Cache Side-channel Attacks

Cache side-channel attacks are possible because (1) the cache is a shared resource by multiple processes and (2) there is a noticeable difference in access time between a cache hit and a cache miss. The specific techniques to attack the cache differs based

[1]The Cortex-A55 technical reference manual states that "L2: Strictly exclusive with L1-D caches [42]." Also, the Cortex-A55 uses the *private per-core* unified L2 cache [42].

[2]The Cortex-A15, 57, 72 technical reference manuals state that "L2: Strictly enforced inclusion property with L1-D caches [25, 34, 35]."

[3]The Cortex-A53 technical reference manual and the other ones *do not* state the inclusiveness of the L2 cache [33, 32, 26, 21, 24, 28, 40, 41, 36, 43].

on the attacker's capabilities in three main areas: whether or not the attacker can reliably control the scheduling of the attack process relative to the victim process, the level of the CPU cache that the attack is targeting, and whether or not the attacker process shares memory with the victim process.

Cache Attack Terminology

In cache side-channel attacks, the attacker uses side-channel information, such as access time, to infer which data has been accessed by the victim. Throughout the paper, I use the term *sensitive data* to denote data that is secret or could be used to infer secrets. For instance, the T-tables of the AES algorithm are sensitive data because the access pattern of T-tables can be used to infer the secret key. Because only a (key-dependent) subset of T-table entries are actually used during encryption, not all of the entries will be put into the cache during an encryption operation. I call the subset of sensitive data that is actually put into the cache during execution *key data*. Thus, the attacker's goal is to use a cache side-channel attack to infer which sensitive data is key data. In addition, I refer to the code that uses sensitive data as *sensitive code*. For example, the implementation of the AES algorithm would be sensitive code.

Attacker Memory Access

Shared memory increases memory efficiency of the system by allowing one copy of the memory contents to be shared by many processes. In addition, this feature enables the system to deploy the cache efficiently. For example, the cache addressing scheme described in Section 2.1 ensures that if one process has loaded data of a shared library, other processes using the same library are able to get the data from the cache quickly [29]. Unfortunately, this feature makes shared libraries *inherently vulnerable*

to cache side-channel attacks by allowing an attacker to measure the loading time of sensitive data in shared memory.

Attack Scheduling

Cache attacks generally include a *setup phase* and a *measurement phase* carried out by the attacker process. Control over the timing of these phases in relation to the execution of sensitive code by a victim process sorts these attacks into two categories: synchronous and asynchronous attacks [133]. *Synchronous attacks* are possible when the attacker is able to schedule the victim process's sensitive code between the setup and measurement phase. By doing so, the attacker can significantly reduce measurement noise and increase the accuracy of the attack. In an *asynchronous attack*, however, the attacker tries to leak information by relying on the expected execution time of the victim process without control over its execution. The accuracy of such attacks, thus, is much lower than that of synchronous ones.

Attacks on Different Cache Levels

Attackers can target different levels of the CPU cache, depending on the specific circumstances of the attack. Traditional techniques targeted the L2 cache, which is shared by multiple cores in a CPU [84, 167, 93, 165]. Thus, side-channel attacks against the L2 cache can be carried out both synchronously and asynchronously.

To attack the L1 cache, attackers have mostly been confined to synchronous, same-core attacks, as this reduces the L1 cache to a shared space between the attacker and victim processes. Recently, however, Irazoqui et al. [78] demonstrated an attack that uses a feature that allows the exchange of cached data between L1 caches (*i.e.*, *directory protocol*). These techniques allow attackers to target the L1 cache even on multi-core systems, and they represent a challenge that SmokeBomb must overcome.

The ARM architecture calls this feature the *AMBA Coherent Interconnect* [38].

2.3 Information Leak Vulnerabilities in OS kernels

For performance concerns, unsafe programming languages, such as C and C++, are still prevalently used in the implementation of operating system (OS) kernels and embedded systems. While these unsafe languages may allocate memory on stack or in the heap for variables, these variables may not be initialized before being used. When a variable is used without proper initialization (which can be caused by either a programming mistake or padding bytes in a struct inserted by compilers [138]), the memory values that were present at the same location of the variable before it was allocated—called *stale values*—will be read and used. When these stale values are copied from the kernel space to the user space, user-space programs will be able to access them, which causes an information-leak vulnerability if the information contained in the stale values is important.

The use of stale values in Linux kernels can lead to severe security problems, which have been studied in the past [54, 116, 97]. Moreover, these stale values can pose severe security threats without being directly used in the kernel. For example, modern kernel security defenses, such as Kernel Address Space Layout Randomization (KASLR), depend on keeping kernel addresses secret from user-space programs. When attackers get lucky and recover kernel pointer values through leaked information (stale values) from the kernel space, they can defeat KASLR [147, 96]. Likewise, attackers may leak cryptographic keys that are stored in the kernel space.

Information Leaks from the Kernel Stack

Each thread in the Linux has a kernel stack, which is a memory area that is allocated in kernel space. Depending on the specific Linux version and configuration, the sizes

```
1  /* file: kernel/time/time.c */
2  COMPAT_SYSCALL_DEFINE1(adjtimex, struct compat_timex __user *, utp)
3  {
4    struct timex txc; //stack object
5    //pass the 'txc' struct without initializing the 'tai' field
6    err = compat_put_timex(utp, &txc);
7    ...
8  /* file: kernel/compat.c */
9  int compat_put_timex(struct compat_timex __user *utp, const struct timex *txc) {
10   struct compat_timex tx32;
11   memset(&tx32, 0, sizeof(struct compat_timex))
12   ...
13   //copy the uninitialized data ('tai')
14   tx32.tai = txc->tai;
15   //kernel data leak to the user-space
16   if(copy_to_user(utp, &tx32, sizeof(struct compat_timex)))
17   ...
```

Listing 2.1: A real-world vulnerability (CVE-2018-11508) in which an uninitialized field of the `time` struct in the stack caused the information leak.

of the kernel stack differ. To maximize the locality, the kernel stacks are usually small in size (8KB or 16KB on x86-64). Therefore, the memory space for the kernel stack is very frequently reused between different kernel stack frames. Lu, *et al.* showed that 90% syscalls only use less than 1,260 bytes of the kernel stack space, and the average stack usage is less than 1,000 bytes [97]. This *reusability* of the kernel stack has resulted in good performance and high efficiency but also has contributed to unexpected leaks of stale values that are left on the kernel stack.

I use two real-world vulnerabilities to demonstrate a kernel information leak vulnerability through uses of uninitialized variables on the stack. Listing 2.1 shows an example of a kernel information leak caused by a use of uninitialized stack memory. In the `adjtimex` syscall, the `txc->tai` field is not initialized and is later used as

```
1  /* file: net/core/rtnetlink.c */
2  static int rtnl_fill_link_ifmap(struct sk_buff *skb, struct net_device *dev)
3  {
4    //all fields in the map object are initialized
5    struct rtnl_link_ifmap map = {
6      .mem_start   = dev->mem_start,
7      .mem_end     = dev->mem_end,
8      .base_addr   = dev->base_addr,
9      .irq         = dev->irq,
10     .dma         = dev->dma,
11     .port        = dev->if_port,
12   };
13
14   //kernel data leak to the user-space
15   if(nla_put(skb, IFLA_MAP, sizeof(map), &map))
16     return -EMSGSIZE;
17   return 0;
18 }
```

Listing 2.2: A real-world vulnerability (CVE-2016-4486) which illustrates that padding bytes inserted by a compiler can bring the information leak.

an argument of `compat_put_timex`. Thus, the `compat_put_timex` function copies the `tai` field of the `txc` object to a local variable (`tx32` object), which is eventually copied to the user space and causes a kernel data leak. Listing 2.2 shows an another example of kernel information leak. Although all fields of the `map` object are initialized by the `rtnl_fill_link_ifmap` function, the object still contains *uninitialized* 4 bytes padding generated by a compiler. Therefore, an unintended kernel data leak occurs when the stack object is copied to the user space by calling the `nla_put` function (on Line 14).

In both examples, the size of leaked data is 4 bytes, which is not enough to fully accommodate an 8-byte pointer value in 64-bit Linux. However, I will show that even a 4-byte leak is sufficient for leaking randomized kernel addresses.

2.4 Temporal Memory Safety Violations

Temporal memory safety violations occur if a pointer is dereferenced when it no longer points to the original object. Such violations help attackers compromise the computer with vulnerable user-space programs and operating systems, as they can be used for privilege escalation or arbitrary code execution. Although temporal memory safety violations lead to various types of vulnerabilities, for ease of illustration, I will consider use-after-free (UAF) vulnerabilities as an example to show the implications of temporal memory violations and motivate the design of our proposed approach.

A UAF vulnerability exists when a pointer value can still be dereferenced after deallocation. To exploit a UAF vulnerability, an attacker must deallocate a *victim object* and create a *dangling pointer*. A *victim object* is a memory object that has been deallocated through a deallocation function (*e.g.*, `free()`), and a *dangling pointer* has a pointer value that points inside a victim object. The attacker will then re-allocate the dereferenced memory region to another object and use the dangling pointer to read from or write to this newly allocated-object, without the constraints that would normally be applied when using the original pointer.

I define prerequisites of the UAF attack in OS kernels as below.

Definition 1. *UAF attacks are possible if and only if there exists a dangling pointer, and performing load/store operations that dereference the dangling pointer do not cause memory access exceptions.*

$$\exists dp : load(dp) \wedge store(dp)$$

Therefore, for a succesful UAF attack, an attacker must make the vulnerability program reallocate the freed memory region with another object that she wants to control through memory collision approaches such as the object-based attack and the

physmap-based attack [145]. Depending on the victim object and other objects that have been placed where the dangling pointer points to, For example, an attacker can read sensitive data, write arbitrary data, or change the kernel's control flows.

Specifically, exploiting a UAF vulnerability requires the following three steps:

(1) Creating a dangling pointer.

(2) Allocating an object to overlap with the deallocated victim object, to which the dangling pointer points.

(3) Dereferencing the dangling pointer.

Hence, to defend against UAF attacks (and any attack that exploits temporal memory safety violations), it suffices to stop the attack at any of these three steps.

Chapter 3

PRIME+COUNT: NOVEL CROSS-WORLD COVERT CHANNELS ON ARM TRUSTZONE

3.1 Introduction

ARM Security Extensions, marketed as TrustZone, have been introduced in ARMv6 and later profile architectures, including Cortex-A (mobile) and Cortex-M (IoT) [23, 22, 30]. The idea of TrustZone is to split the system-on-chip hardware and software into two security states or worlds, namely *normal world* and *secure world.* Hardware barriers are established to prevent normal world components from accessing secure world resources. However, the secure world components are not restricted to access normal world resources.

Two legitimate channels exist at the hardware level that a normal world component and a secure world component can use to communicate with each other. The first channel is that either world can put messages in the general registers when a world switching is performed. The second channel is the secure world can directly read and write to a region of physical memory that normal world can also access. These two channels can be used together for faster communication: a normal world component invokes the Secure Monitor Call (`SMC`) instruction with general registers containing the address of allocated memory in the normal world. Then, the normal world and secure world components can send requests and get responses using the shared memory.

Previous studies have shown that these legitimate channels are vulnerable to an attacker who has the normal world kernel privileges and keeps sending crafted argu-

ments to probe the vulnerabilities of the secure world [104, 121, 80, 86]. For example, Android full disk encryption was broken by sending crafted messages in these channels and eventually gaining code execution privileges within the secure world kernel [86]. There are two ways to protect these channels from being abused:

(1) Prior work, SeCReT [80], has aimed at restricting the access to the communication channels and secure world resources to normal world components on an access control list (ACL). SeCReT ensures only predefined and legitimate normal world components can communicate and access secure world resources. To this end, SeCReT authenticates a normal world component by verifying its code and control integrity when it initiates communication with secure world. Consequently, unauthenticated normal world components cannot access the cross-world communication channels.

(2) It is possible to deploy a strong monitor, similar to a network intrusion detection or deep packet inspection system, in legitimate communication channels, including parameters passed by registers and shared memory, between the normal and secure world to inspect all transmitted data and block illegal communication when it is detected. Even though how to design such strong monitors is a research problem itself, and no practical solutions exist to the best of our knowledge, we assume that they could exist in the future.

In this work, we are interested in building cross-world covert channels in the TrustZone architecture that (1) enable unauthenticated normal world and secure world components to communicate even when solutions like SeCReT are deployed; (2) enable normal world and secure world components to communicate even when strong monitors that can inspect all transmitted data in legitimate channels are deployed in the future. As a result, a secure world component can always smuggle sensitive information that is not supposed to leave the secure world to the normal world, such as private keys, user passwords, etc. And, a normal world component can send secret

messages (e.g., command and control messages) to secure world.

The emergence of downloadable Trusted Applications (TAs) gives such covert channels even more practical use-scenarios [150], where a malicious TA can steal sensitive information that does not belong to it in the secure world and send to its counterpart in the normal world, hence circumventing SeCReT and strong monitor.

We propose to build covert channels using a trade-off between performance and cost in the TrustZone hardware, which are not governed by any software solution built on top of TrustZone, such as SeCReT or strong monitors. We notice that even though many system-on-chip resources are separated in the TrustZone architecture, there is only one copy of cache in the system that is shared between the two worlds. In each cache line, there is one extended bit to indicate whether the memory content is from a secure or normal world memory region. This simple extension can prevent normal world components from accessing cache contents of secure world, but sharing the cache between the normal and secure world itself makes some cache-based attacks possible [91, 154].

Even though it is easy to assume covert channels must exist given that cache is shared between worlds, there is no comprehensive study on the practicality and bandwidth of cross-world covert channels in the TrustZone architecture. In this work, we identify several challenges in building such cross-world covert channels: (1) the pseudo-random replacement policy on ARM makes Prime+Probe less reliable [91]; (2) the cross-world context switching also introduces much noise. Our work confirms that Prime+Probe is not reliable in the cross-world scenario; (3) low noise and fine-grained cache line-level attacks, such as Flush+Reload [157] and Flush+Flush [68], require sharing memory objects between the *Sender* and the *Receiver*, which does not fit in a practical attack model.

To cope with these challenges, we need a novel cache attack approach that does

not require memory sharing and introduces less noise in the cross-world scenario. We leverage an overlooked ARM Performance Monitor Unit (PMU) feature named "L1/L2 cache refill events" and design a PRIME+COUNT technique that only cares about *how many* cache *sets* or *lines* have been occupied instead of determining *which* cache *sets* have been occupied as in PRIME+PROBE. The coarser-grained approach significantly reduces the noise introduced by the pseudo-random replacement policy and world switching. Even though some performance counters in PMU, such as cycle counter, have been used to carry out and detect cache-based side-channel attacks in the ARM and Intel architecture [154, 53], to the best of our knowledge it is novel to use "L1/L2 cache refill events" to perform attacks.

We leverage the PRIME+COUNT technique to build covert channels in single-core and cross-core scenarios in the TrustZone architecture. To evaluate the efficacy of the covert channels, we test the implementations on two devices, one of which is a Samsung Tizen TV with ARMv7 CPU and the other is a Hikey board with ARMv8 CPU. The evaluation results show that the bandwidth could be as high as 27 KB/s in the single-core scenario and 95 B/s in the cross-core scenario.

3.2 ARM TrustZone

In this section, we first present an overview of the ARM TrustZone architecture. Then, we elaborate on the legitimate channels between the normal and secure world.

3.2.1 ARM Architecture and TrustZone

Processor Modes

An ARM processor has up to 9 different modes depending on if some optional extensions have been implemented. The user (`usr`) mode has a privilege level 0 and is where user space programs run. The supervisor (`svc`) mode has a privilege level 1

and is where most parts of kernel execute.

TrustZone and Processor States

TrustZone is a hardware security extension of the ARM processor architecture, which includes bus fabric and system peripherals. When TrustZone is implemented, a processor has two security states or worlds, namely the secure world (`s`) and the normal world (`ns`). The distinction between the two states is orthogonal to the processor modes. The partitioning of all the System on Chip (SoC)'s hardware and software into two worlds may be physical and/or virtual. For instance, a processor core is shared by the normal and secure world in a time-sliced fashion. World switching is done in the monitor mode after calling the secure monitor call (SMC) instruction in either world. The SMC instruction forces the running core to enter the monitor mode.

The secure world enables the construction of an isolated programmable environment that can run a wide range of security applications. The secure world is everything runs when the processor state is secure, and normal world is everything runs when the processor is in the non-secure state. Hardware barriers are established to prevent normal world components from accessing secure world resources; the secure world is not restricted. Specifically, the memory system prevents the normal world from accessing (1) regions of the physical memory designated as secure; (2) system controls that apply to the secure world; and (3) state switching outside of a small number of approved mechanisms.

3.2.2 Legitimate Channels Between the Normal and Secure Worlds

At the hardware layer, there are two ways for a normal world and a secure world component to communicate with each other. Firstly, messages can be stored in the

general registers when a world switching happens, which is triggered by the `SMC` instruction. For instance, secure monitor call calling convention [37] defines how parameters are passed through the general registers, and it is implemented in firmware, such as ARM Trusted Firmware [39]. Previous projects, such as SeCReT [80], attempted to add extra layers of authentication and verification to make sure only predefined and legitimate components can use this channel.

Secondly, the secure world kernel can directly map a memory region that is accessible by the normal world. Hence, this shared memory region can be used by the normal and secure world to communicate. Secure world OSes, such as OP-TEE [110], have implemented shared memory. Usually, a physical memory region is first allocated by the normal world kernel. The physical address, the size of the shared memory, and other important information are then transferred to the secure world OS through the `SMC` interface, so the secure world can configure its MMU table entries to access the region directly. Since important information is still passed through the `SMC` interface, solutions such as SeCReT can also monitor this channel. Besides SeCReT, we can assume strong monitors can be implemented in the future to inspect all transmitted data in these channels.

3.2.3 Cache in TrustZone Architecture

Unlike some banked system registers, there is only one copy of cache that is shared between normal and secure world. Each cache line has one bit to indicate if its content is from a secure or normal world memory region. Even though this extend bit can prevent normal world components from accessing cache contents of secure world, the design of shared cache still makes some cross-world cache attacks possible.

3.3 Assumptions and Attack Model

We assume a solution, such as SeCReT [80], that only allows authenticated normal world components to use the communication channel, is running in secure world monitor mode. Such a solution safely maintains a list of predefined normal world components that are allowed to use the legitimate channels. We also assume that there is a strong monitor that can understand all transmitted data between the normal and secure world and block illegal communications.

The goal of an attacker is to smuggle sensitive information that is only accessible in the secure world to the normal world. To this end, the attacker runs a component, namely *Receiver* in the normal world and another component, namely *Sender* in the secure world. Because legitimate communication channels, including parameters passed by registers and shared memory, between the normal world and the secure world are under inspection by a SeCReT or a strong monitor, it is impossible for the *Sender* and the *Receiver* to transfer sensitive data from the secure to the normal world using such channels without being detected. To bypass this kind of cross-world communication monitoring, the *Sender* and the *Receiver* need to use channels that are not governed by the sentries implemented in the monitor mode.

We also assume the attacker has kernel privileges in the normal world, so the *Receiver* can use privileged instructions to access the PMU. This constraint can be loosened if the *perf_event_open* system call is provided to monitor "L1/L2 cache refill events" in userland. The *Sender* can simply be a secure world application (trusted application), and it is not necessary for it to have kernel privileges. This is because the *Sender* will only need to influence cache by reading/writing memory regions and does not need to access the PMU. However, having the *Sender* running in the kernel space enables it to steal information that is not available for userland processes.

Running an application in the secure world is very feasible for the attacker who can either leverage vulnerabilities of the secure world interfaces as shown in [87, 107] or bypass application vetting mechanisms [51]. The use of downloadable TAs, which are predicted to be used widely [150], would increase the chance as well.

In summary, in the attack model, attackers are not stronger than their counterparts in previous events [104, 121] or in the attack model presented in SeCReT [80], except that the *Sender*, which can be a userland application, is a must. My implementation suggests such an application could be implemented in hundreds of lines of C code. Moreover, PRIME+COUNT attack can be carried out even when mechanisms, such as strong monitors, that are more powerful than normal world component authentication, such as SeCReT, are deployed between the two worlds.

Depending on the hardware the attack is performed on and resources the attacker possesses, we articulate two attack scenarios: single-core and cross-core.

(1) *Single-core scenario*: This scenario occurs when either the targeted device only has a single-core CPU or the attacker can only control one of the cores in a multi-core CPU. Because there is only one core available to the attacker, the attacker needs to use the `SMC` instruction to switch between the normal and secure world. In addition, in this scenario the attacker can use either L1 cache or L2 cache. Note that even if the attacker can use the `SMC` instruction in this scenario, it is not possible to send sensitive information directly using the `SMC` instruction or shared memory due to the sentry in the monitor mode;

(2) *Cross-core scenario*: In this scenario, the attacker can execute the *Receiver* in the normal world and the *Sender* in the secure world on two different cores at the same time. Because different cores do not share L1 cache, the covert channel can only be constructed using the L2 cache. Therefore, the inclusiveness of L2 cache affects the result. In this scenario, there is no need for the attacker to use the `SMC` instruction

Device	SoC	CPU (cores)	L1 Cache	L2 Cache	Inclusiveness	Secure World OS	Normal World OS
Samsung Tizen TV	N/A	32-bit Cortex-A17 (4)	32KB, 4-way, 128 sets	1024KB, 16-way, 1024 sets	Inclusive	`SecureOS`	Tizen OS on Linux 4.1.10
Hikey board	HiSilicon Kirin 620	64-bit Cortex-A53 (8)	32KB, 4-way, 128 sets	512KB, 16-way, 512 sets	Inclusive	ARM Trusted Firmware, OP-TEE	Linux 4.1.0

Table 3.1: Test Environments.

to switch between the worlds.

In this work, we attempt to solve the challenges in building cross-world covert channels in both aforementioned scenarios. All experiments are performed on the two environments as listed in Table 3.1. In addition, we use a TRACE32 hardware debugger[1] to trace cache operations on the Tizen TV.

3.4 Cross-world Covert Channels

In this work, we propose the covert channel between the two worlds using cache, which can defeat the fundamental security feature of the TrustZone. The goal of the covert channel is to construct pragmatic data channel between two worlds secretly.

At a high level, to build cache-based covert channels, the *Receiver* first makes the whole cache or some specific cache lines enter a known state. To this end, the *Receiver* can fill the cache with contents from its own address space. In the second step, the *Sender* carefully changes states of some cache lines by evicting the contents of those lines and placing its own contents there. In the third step, the *Receiver* detects such changes to decipher the message the *Sender* transmits. Note that, in almost all the platforms, neither the *Sender* nor the *Receiver* can directly read the content of any cache line. Therefore, the message is actually delivered using channels such as *which* specific cache lines or sets have been changed in previous projects [120, 144, 91]. To

[1]http://www.lauterbach.com/

Algorithm 1: PRIME+COUNT-based Cross-world Covert Channels: PRIME part. x is the message to be sent.

```
   /* Receiver:  Prime                                              */
 1 if Single-core covert channel then
 2     for Each L1-D cache line do
 3         Clean & Invalidate the L1-D cache line
 4         Load data to fill the L1-D cache line
 5     Yield control to the secure world by executing SMC
 6 if Cross-core covert channel then
 7     for Each L2 cache line do
 8         Clean & Invalidate the L1-D cache line
 9         Clean & Invalidate the L2 cache line
10         Load data to fill the L1-D & L2 cache lines
11     Clean & Invalidate the whole L1-D cache
```

receive such information, the *Receiver* accesses its own address space again and uses cache hit or miss to detect how many cache lines have been changed.

Our approach follows this general idea with some changes that are tailor-made for the TrustZone architecture. In particular, we propose PRIME+COUNT, it uses *the number* of changed cache lines as the covert channel instead of *which* cache lines or sets. Algorithm 1 and 2 demonstrates the overall workflow of building cross-world covert channels using PRIME+COUNT. As shown in Lines 2–4 and 7–11 in Algorithm 1, the *Receiver* first PRIMEs the cache. Because covert channels are based on the number of cache misses, the results of the PRIME step can have a strong influence on the reliability and bandwidth of the covert channel. Due to the pseudo-

Algorithm 2: PRIME+COUNT-based Cross-world Covert Channels: COUNT part. x is the message to be sent.

```
   /* Sender:  Write to covert channel                          */
1  if Single-core covert channel then
2      Occupy x L1-D cache lines
3      Yield control to the normal world by executing SMC
4  if Cross-core covert channel then
5      Occupy x L2 cache lines
   /* Receiver:  Count                                          */
6  Determine how many cache lines are changed by Sender
7  Apply bucket method for further noise reduction
```

random cache replacement policy, an effective and efficient PRIME method is not very straightforward. We discuss the PRIME method in detail in Section 3.4.2.

In the single-core scenario, the *Receiver* then needs to yield control to the secure world so the *Sender* can execute as shown in Line 5. In the cross-core scenario, this step is omitted. Then, as shown in Lines 1 and 5 of Algorithm 2, the *Sender* writes data to the covert channel by occupying x cache lines, where x is the message to be sent. In this step, the cache replacement policy could be the obstacle again. Consequently, a similar method in PRIME is used for accurate message writing. In the single-core scenario, the *Sender* then yields control to the normal world so that the *Receiver* can decode the message. Lastly, the *Receiver* COUNTs how many cache lines are changed as shown in Line 6 and uses a simple noise reduction method to get the message as shown in Line 7.

The main difference in single-core and cross-core scenario is that the L2 cache is

used in the cross-core scenario instead of the L1-D cache. We discuss the details of the differences in Section 3.4.5.

3.4.1 Prime+Count *Overview*

We first explain why Prime+Probe is not the best choice in cross-world scenario. Then, we illustrate how Prime+Count works and the challenges we have solved in designing it.

Why not Prime+Probe?

Intuitively, Prime+Probe can be used to build cross-world covert channels in our attack model. It is not the best option due to the following reasons:

(1) *Noisy:* Due to ARM's pseudo-random replacement policy, Lipp et al. demonstrated that Prime+Probe is not reliable [91]. The world switching introduced by TrustZone increases the ineffectiveness of Prime+Probe. In addition, during the time when the normal world part of the covert channel is working, other kernel code executing on the same core can introduce extra noise.

We conducted several experiments on both devices to show how much noise can be introduced on each set of the L1-D cache during the world switching after the Prime. In the experiments, the secure world simply yields control to the normal world after loading a specific number of cache sets. Figure 3.1 shows how many cache misses occurred for each cache set in 200 world switchings on the Hikey board. The x-axis is the index of the cache sets from 0 to 127, and the y-axis is the accumulated number of cache misses. The experiments suggest the noise is widely dispersed on the cache sets and the average number of cache misses per world switching is around 18 over 128 cache sets. Even though Figure 3.1 shows some cache sets, such as cache set 1, are never used during the world switching in our experiments on the Hikey board, it does

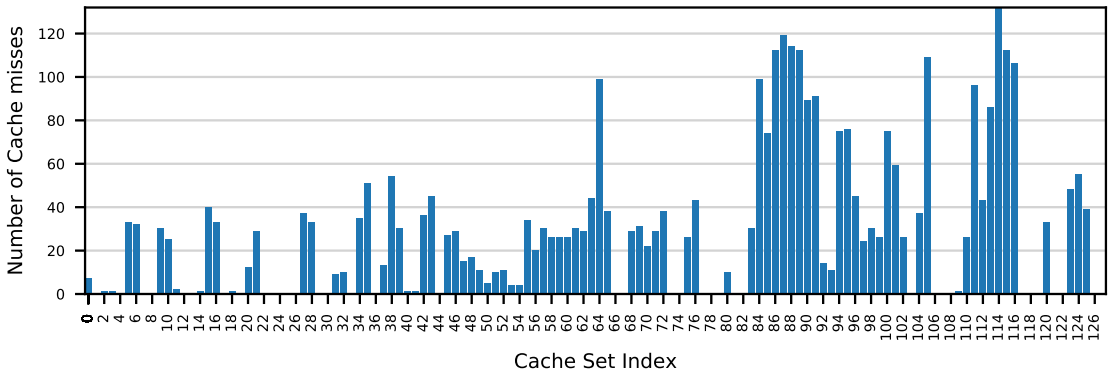


Figure 3.1: Cache Misses Introduced by World Switching.

not mean that those cache sets are guaranteed to stay intact when other hardware devices or different firmware and OS are used. Hence, it is not feasible to use this observation to build generic covert channels for a variety of hardware and software environments.

(2) *Difficult to choose threshold:* In the ARM architecture, Performance Monitor Unit (PMU) is implemented to provide statistics of CPU activities. In particular, the Performance Monitors Cycle Count Register (PMCCNTR) that increments from the hardware processor clock. By subtracting a previously recorded PMCCNTR value (p_1) from its current value (p_2), the number of elapsed processor cycles ($\Delta = p_2 - p_1$) can be easily computed. To distinguish between cache hit and miss for a memory access, PMCCNTR is read before and after the memory access attempt. If the number of elapsed cycles is greater than some predefined threshold ($\Delta > \theta$), the attempt is classified as a cache miss; otherwise, it is considered as a cache hit. This approach has been used in Prime+Probe and other cache attacks. However, *the thresholds used for decision making are contingent upon the implementation of the CPU*, which means there is no one-size-fits-all threshold value for all available devices on the market. Even though Lipp et al. proposed a mechanism to automatically compute

the threshold at run-time [91], it inevitably increases the size of the attack code base and the chance to be discovered.

Why Prime+Count?

PRIME+COUNT counts *how many* cache *sets* or *lines* have been occupied instead of determining *which* cache *sets* have been occupied. PRIME+COUNT, as a coarser-grained approach than PRIME+PROBE, significantly reduces the noise introduced by the random replacement policy and world switching. In addition, PRIME+COUNT does not require shared memory space or shared memory objects with a victim. PRIME+COUNT only cares about how many cache sets/lines have been changed. Therefore, it may be difficult to use it for some attacks other than building covert channels, such as stealing cryptographic algorithm keys.

3.4.2 PRIME *the Cache*

Ineffective PRIME affects the accuracy of COUNT and adds noise to the covert channel. It is suggested that the pseudo-random cache replacement policy is a significant obstacle in PRIME [154, 91]. Taking a 4-way set associative cache as an example, based on the index of the physical address newly fetched data can be loaded to any of the 4 ways. Therefore, even if we load as much data as the size of the L1-D cache, there is no guarantee that the cache will be completely occupied.

Previous Prime Method

Previous approaches to achieve high cache coverage in PRIME for userland programs load data repeatedly using various access patterns [154, 91]. However, this type of approach costs thousands of CPU cycles even when it is only used to prime a small portion of the cache [91].

Message to be Sent	# of Cache Sets should be Occupied by the *Sender*	# of Cache Events detected by the *Receiver*	Bucket Ranges set by the *Receiver*	Message decoded by the *Receiver*
0	10	23	0 - 43	0
1	40	60	44 - 71	1
2	70	85	72 - 99	2
3	100	111	100 - 128	3

Table 3.2: An Example of the Bucket Method: We assume there are 128 sets and set-counting mode is used. The channel can transmit as most 7 bits ($log_2 128$) every time. In this example, only 2 bits are transmitted.

Also, our experiments confirm that repeating the data loading at kernel level is costly. We perform a systematic analysis using the TRACE32 hardware debugger to dump the content of cache on a Samsung Tizen TV. To this end, we prepare 32 KB of memory space, the same size of the L1-D cache. Then, we access the first byte of the memory and keep accessing data at the address that is 64 bytes (size of cache line) away from the one before. After repeating this operation for 512 times (128 sets $\times$ 4 ways), we dump the content of cache using the TRACE32 debugger. To minimize possible interference, we use a spinlock to give our experiment code exclusive use of the core. Our results show that, on average, only 372 of 512 cache lines were occupied after accessing the 32 KB memory once. Only by repeating this procedure for more than 50 times could it achieve around 95% cache occupation.

Our Prime Method

Obviously, a faster PRIME method could significantly increase the bandwidth of covert channels and reduce the chance of being discovered. In this work, as shown in Lines 3–4 in Algorithm 1 we clean and invalidate each cache line before only loading the data to cache once. Our experiments show that this method achieves around 99% occupation

Algorithm 3: An Alternative Method.

```
   /* The following lines replace Lines 2-4 in Algorithm 1       */
1  for Each L1-D cache set do
2      Clean & Invalidate the L1-D cache set
3  for Each L1-D cache set do
4      Load data to fill the L1-D cache set
```

on average.

This method operates as follow: (1) The starting address of a memory block is assigned to the pointer; (2) we translates virtual address to physical address. Once the physical address is obtained, we can extract its *set* number; (3) After that, we select the target cache line among the lines (ways) in the set using the DC CISW instruction. The DC CISW instruction's operands are a *set* number and a *way* number, and thus, we can choose a specific cache line (way) in a set to clean and invalidate it. We typically start from the way 0 to the last way; (4) Lastly, we load the data to the cache line. The pointer is increased by the length of a cache line so that we can point to the next cache set of the way in the next round. If the *way* has been fully filled by data, we fetch data to the next way. Steps (1) – (4) are iterated until PRIME is done.

We also conduct experiments with an alternative method shown in Algorithm 3. In this method, we clean and invalidate all cache lines of the L1-D cache before loading the data. Experiments show that this method achieves around 95% occupation on average.

3.4.3 COUNT *Using Cache Refill Events*

The Performance Monitor Unit (PMU) includes logic to gather various statistics on the operations of the processor and memory system during runtime. We use overlooked PMU features called "L1/L2 Cache Refill Event" to count how many cache lines have been updated. A cache refill event can be triggered by any access causing data to be fetched from outside the cache. Therefore, every cache miss can be counted by using the event.

After the secure world occupies some cache lines using the PRIME method, it yields control to normal world, and COUNT function will execute. If a cache line is refilled while accessing the memory, the counter will increment. Therefore, this function gives us how many cache lines have been changed between PRIME and COUNT.

Two Counting Modes

There are two counting modes we use in the experiments:

Line-counting mode. The smallest unit for counting a cache refill event is a line. For example, if the L1-D cache is a 128-set 4-way cache, we can check each of the 512 lines to count how many refill events occur. In this mode, the covert channel can transmit at most 9 bits ($log_2 512$) every time.

Set-counting mode. Another option is to count the cache refill events on only one way, so just 128 lines will be checked. A way can be chosen by using the DC CISW instruction. The former, which will be referred to as 'line-counting mode' in this work, can send 9 bits per time ideally with the PRIME+COUNT, In this mode, the covert channel can transmit at most 7 bits ($log_2 128$) every time. However, we only need to PRIME one way in this mode. Therefore, this mode can achieve higher bandwidth than the line-counting mode.

Defeating Data Prefetching

One of the challenges we encountered in implementing COUNT is the automatic data prefetcher [22, 30]. Data prefetching is a technique that fetches data into the cache earlier than the instruction that uses the data is executed. To do so, the prefetcher monitors data cache misses and learns an access pattern. However, a data prefetching does not trigger a refill event. So, the counter will not increment when a new cache line fill is caused by data prefetching.

There are several methods to prevent data prefetching. One way is to disable the prefetcher directly by changing the corresponding bit in the auxiliary control register. However, it is only safe to do so after the MMU is enabled, which does not fit in our attack model. Moreover, disabling the prefetcher will downgrade the performance of the system. Another way is to access memory locations in a random and unpredictable order, so it is difficult for the prefetcher to learn a pattern. However, this method increases the complexity of implementing covert channels.

We solve the problem by employing the instruction synchronization barrier (`ISB`). The `ISB` instruction flushes the pipeline of a core and the prefetcher buffer as well. It is normally used when the context or system registers are changed as well as after the branch predict maintenance operations.

3.4.4 A Simple Message Encoding Method

Even though PRIME+COUNT introduces significantly less noise than PRIME+PROBE, noise is still inevitable due to the world switching and other factors. One way to correct the errors introduced by noise is to adopt error correction encoding methods, such as Reed-Solomon error correction [118]. However, those encoding methods significantly (1) increase the size of message, (2) are time consuming to perform, and

(3) increase the size of the code base. Hence, adopting those methods could even further reduce the bandwidth of the covert channel and increase the chances of being discovered. A recent study also suggests that directly applying error-correcting codes does not work due to cache-based covert channel noise characteristics [99].

Fortunately, our empirical experiments show that the introduced noise in PRIME+COUNT (error in number of cache refill events) is manageable. Therefore, we design a simple encoding method, which essentially ignores the least significant bits of the received data. We call this approach the *bucket method*.

The basic idea of the bucket method is to divide the numbers of cache refill events into several groups. Table 3.2 illustrates one example of using the bucket method when 2 bits of data are transferred from the secure world using a 7-bit channel (128 sets in set-counting mode). In this example, when the *Sender* wants to send message 2, it will try to occupy 70 cache lines, which may result in 85 cache refill events detected by the *Receiver*. The *Receiver* then uses the bucket method to decode the message back to 2. The range of a bucket should be decided empirically.

3.4.5 Cross-Core Covert Channels

We use the same PRIME+COUNT approaches as the single-core covert channel for cross-core covert channel except for the level of the cache refill event. Besides that, as shown in Algorithm 1 Line 11, the whole L1-D cache should be cleaned and invalidated after the PRIME. If the L1-D cache has data which was used to occupy the L2 cache after the PRIME, the remaining data in the L1-D cache will cause cache hits during COUNT even if the secure world the *Sender* loads all cache lines of the L2 cache. Cleaning and invalidating the L1-D cache using the `DC CISW` instruction does not affect the L2 cache.

Because the L2 cache is shared by many cores and the cache size is much bigger

than the L1-D cache, in practice it is impossible to prevent other cores from changing the cache lines during the time of PRIME or after PRIME. Therefore, the noise caused by other cores makes line-counting mode infeasible for building cross-core covert channels. Consequently, we design a modified set-counting mode. The set-counting mode for the single-core environment counts cache misses of one way. For the cross-core covert channel, we check cache misses of all cache lines in a set spanning all ways.

3.4.6 Noise of the Covert Channel

Unlike the other previous covert channels, we aimed at the cross-world covert channel which is along with the path of world switching. Thus, transferred data using the covert channel has a noise as it moves through the path. We defined a noise as an error that cause unexpected cache misses or cache hits, which disturbs accurate data transfer. As far as the goal of this work is implementing practical and covert data channel, we need to clearly figure out what types of the noise are produced and how can we deal with that.

We empirically analyzed the noise and propose our approach based on the analysis results. There are 2 types of the noise in the channel, which are classified according to the effects. The first type results in additional cache misses. The second type issues the cache hit at the address where the cache miss should be happened.

Basic reason with respect to the first type noise is data used during the world switching and by functions that the channel have to be gone through. In addition, the kernel codes can make another noise. Even though we used the spinlock to being not yielded to other process, the locks cannot guarantee that the kernel codes will not working on the core while the normal world part of the covert channel is working. These noise are inevitable unless we changed the system. Also, we cannot calculate

how many noise will be produced during the world switching since the pseudo-random replacement policy.

Inaccurate prime method can be another source of the first type noise. When we check the cache misses, we access all data which were used to do the PRIME again. Once the data is remained in the cache, the cache refill event will not be occurred, otherwise, the event counter will be increased. Therefore it is possible that un-fetched data in the RAM can be fetched into the cache during the COUNT if the normal world cannot occupy the L1-D cache totally.

In case of the second type noise, one fundamental source is also the uncertainty of the prime method. As far as the prime method is not able to fill the cache completely, the number of cache misses, can be lower than the number we intended. When the secure world cannot occupy the cache lines completely, there will be less number of cache miss than the lines that the secure word accessed with the consequence that. The second type noise, also, can be occurred by the data prefetcher.

Accordingly, there are noise that we have to embrace unavoidably but the noise caused by the inaccurate prime method and the data prefetcher is manageable. In this work, we prove how to reduce the noise significantly with the PRIME method and by deterring the automatic data prefetcher.

3.5 Implementation

We implemented the PRIME+COUNT method and covert channels on the two devices as listed in Table 3.1. Also, we open source the prototype[2] with the expectation that it will be utilzed and extened by security researchers.

The software implementation consists of a normal world module (the *Receiver*) and a secure world module (the *Sender*) to simulate the scenario that the *Sender*

[2] `https://github.com/Samsung/prime-count`

tries to smuggle sensitive data out to the normal world. Note that with a simple implementation twist the PRIME+COUNT technique and covert channels based on it can be used to send data from the normal world to the secure world as well.

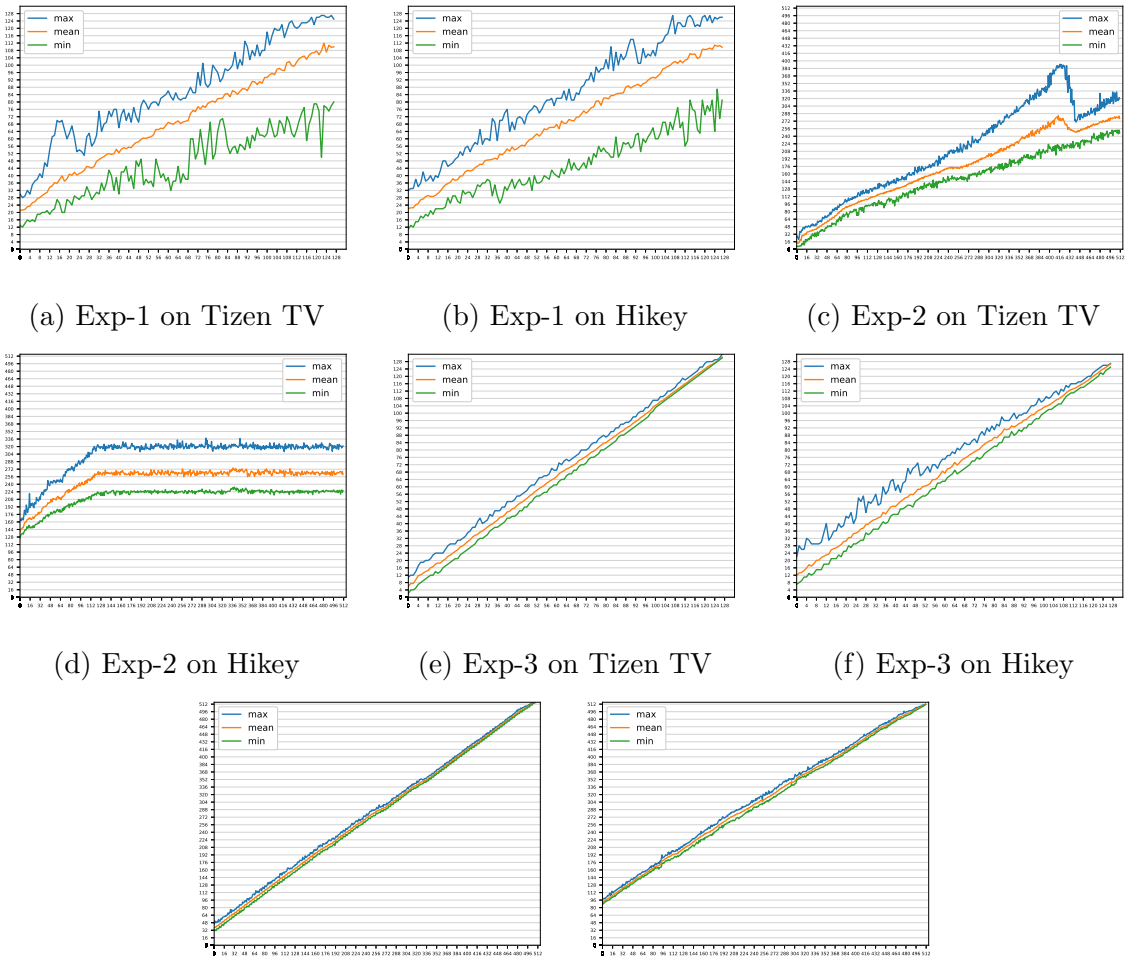
In the single-core scenario implementation, the normal world module is a loadable kernel module (LKM) that can execute the `SMC` instruction directly. In the case of the Samsung Tizen TV, the *Sender* is a secure world application that does not have kernel privileges. To invoke the application, a new `SMC` handler is added to the kernel of the `secureOS`. Note that, in Samsung Tizen TV, only authenticated trusted applications can be loaded on the `secureOS`, and only Root TA can load LKM to the kernel of `secureOS`. Therefore, in practice a malicious kernel-level *Sender* needs to bypass Samsung's code vetting first. For the Hikey board implementation, we implemented the *Sender* in kernel by modifying the `tee_entry_fast` function in the `entry_fast.c` of the OP-TEE.

In the multi-core covert channel scenario, we implemented two kernel threads in the normal world and assigned each of them to a different physical core. One of the threads acting as the *Receiver* stays in the normal world. The other kernel thread executes `SMC` and invokes the *Sender* in the secure world. The *Sender* and the *Receiver* use L2 cache to communicate.

The normal world kernel module consists of 1,134 SLoC for both test environments. The secure world implementation on the Hikey board consist of 84 SLoC, whereas the secure world application on Samsung Tizen TV has 319 SLoC.

3.6 Evaluation

In this section, we report the evaluation results of cross-world PRIME+COUNT-based covert channels on the TrustZone architecture. In section 3.6.1, we evaluate how much noise our PRIME+COUNT method could reduce compared with previous



(a) Exp-1 on Tizen TV (b) Exp-1 on Hikey (c) Exp-2 on Tizen TV

(d) Exp-2 on Hikey (e) Exp-3 on Tizen TV (f) Exp-3 on Hikey

(g) Exp-4 on Tizen TV (h) Exp-4 on Hikey

Figure 3.2: The Effectiveness of Our PRIME Method: We conducted multiple experiments on both devices. Exp-1: PRIME with repeated loading, no instruction barrier, set-counting mode; Exp-2: PRIME with repeated loading, no instruction barrier, line-counting mode; Exp-3: Our PRIME method, with instruction barrier, set-counting mode; Exp-4: Our PRIME method, with instruction barrier, line-counting mode.

approaches. Section 3.6.2 discusses how we choose bucket ranges in the experiments. Section 3.6.3 measures the bandwidth of covert channels under different conditions. In Section 3.6.4 shows images transferred using covert channels.

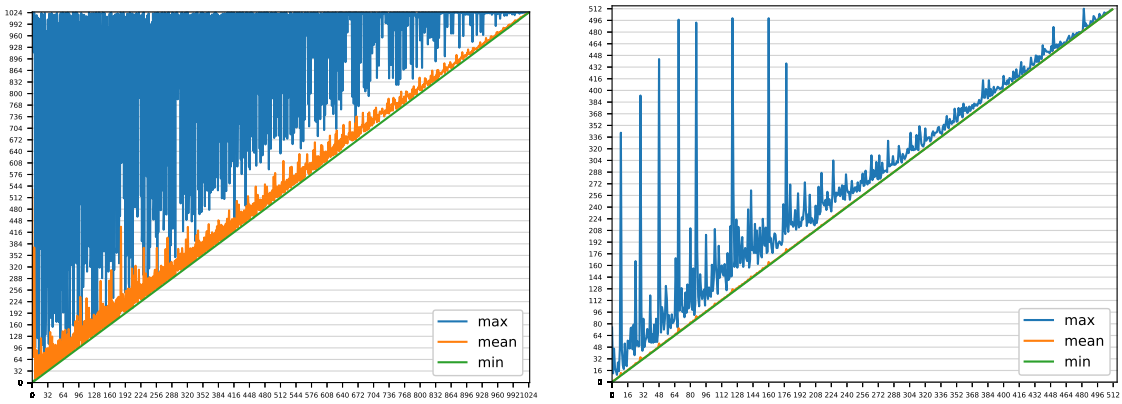
3.6.1 Effectiveness of PRIME+COUNT

Single-core Scenario

We designed four experiments to demonstrate the effectiveness of our PRIME+COUNT method under the single-core scenario. In each experiment, the *Sender* tries to load a specific number of lines/sets (x-axis), and the *Receiver* detects how many lines/sets changes (y-axis). The configurations of experiments are listed as follows: (1) Exp-1: PRIME with repeated loading 50 times, no instruction barrier, set-counting mode; (2) Exp-2: PRIME with repeated loading 50 times, no instruction barrier, line-counting mode; (3) Exp-3: Our PRIME method, with instruction barrier, set-counting mode; (4) Exp-4: Our PRIME method, with instruction barrier, line-counting mode.

We repeated the experiment on each device 1,000 times. Figure 3.2 shows the evaluation results. The x-axis represents the number of cache lines/sets loaded by the secure world *Sender*, whereas the y-axis represents how many L1 cache refill events were detected by the *Receiver*. The blue line indicates the maximum number of cache fill events detected, whereas the green line shows the minimum number of cache fill events detected. The orange line denotes the average over the 1,000 experiments.

From the first row of Figure 3.2, which is the previous PRIME approach on both devices, We can see those approaches are far from reliable and the gaps between the maximums and minimums are large. It is particularly interesting to see the number of cache refill events will even stay at around 256 on average no matter how many lines the *Sender* tries to load as shown in Figure 3.2-(d). We tried to look for



(a) Exp-5 on Tizen TV

(b) Exp-5 on Hikey

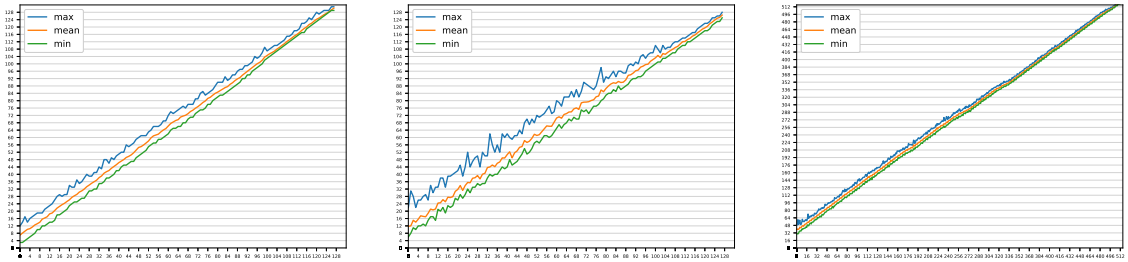
Figure 3.3: Number of Loaded Cache Sets Versus Detected L2 Refill Events under the Cross-core Scenario.

Attack Scenario	Test Device	Counting Mode	Exp-10	Exp-11	Exp-12	Exp-13
Single-core	Samsung Tizen TV (Cortex-A17)	Set-counting	10,330.97	27,408.13	4,868.28	12,971.03
		Line-counting	5,293.62	8,216.97	2,517.62	3,892.50
	Hikey Board (Cortex-A53)	Set-counting	10,273.43	15,646.21	3,812.29	6,201.89
		Line-counting	2,605.33	5,101.91	875.12	1,824.15
Cross-core	Samsung Tizen TV (Cortex-A17)	Set-counting	19.32	45.83	15.31	17.73
	Hikey Board (Cortex-A53)		52.14	95.04	22.33	26.49

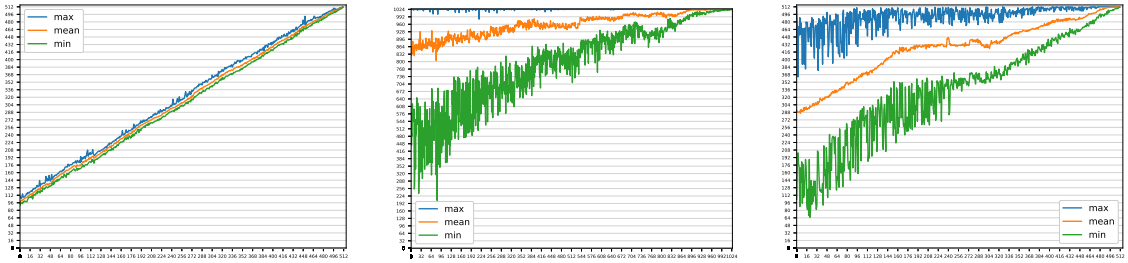
Table 3.3: Capacities of Covert Channels (Byte/Second).

explanations and failed to find any answers in any official specifications of Hikey or ARM documents.

By comparing the first row (previous PRIME techniques) and the second row (our PRIME technique) of Figure 3.2, we can clearly see that the variance of noise is significantly reduced using our PRIME method with an instruction barrier.



(a) Exp-6 on Tizen TV (b) Exp-6 on Hikey (c) Exp-7 on Tizen TV

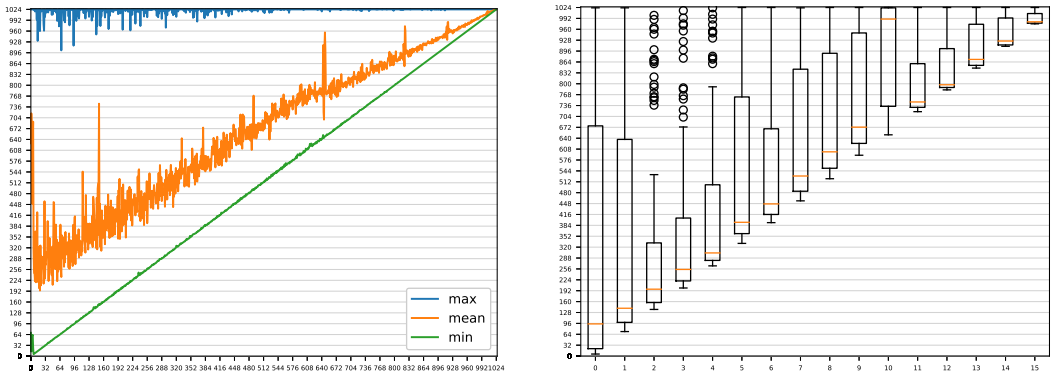


(d) Exp-7 on Hikey (e) Exp-8 on Tizen TV (f) Exp-8 on Hikey

Figure 3.4: Number of Loaded Cache Sets Versus Detected Refill Events under Extreme Conditions: We conducted multiple experiments. (1) Exp-6: the set-counting mode under the single-core scenario; (2) Exp-7: the line-counting mode under the single-core scenario; (3) Exp-8: the set-counting mode under the cross-core scenario.

Cross-core Scenario

We also conducted cross-core experiments on both devices using our PRIME method, with instruction barrier and set-counting mode (Exp-5). As Figure 3.3 shows, the noise under the cross-core scenario is much stronger than it is under the single-core scenario. Also, the results on Hikey is more stable than the results on Tizen TV. This is because there are several applications running on the Tizen system when we were conducting the experiments.



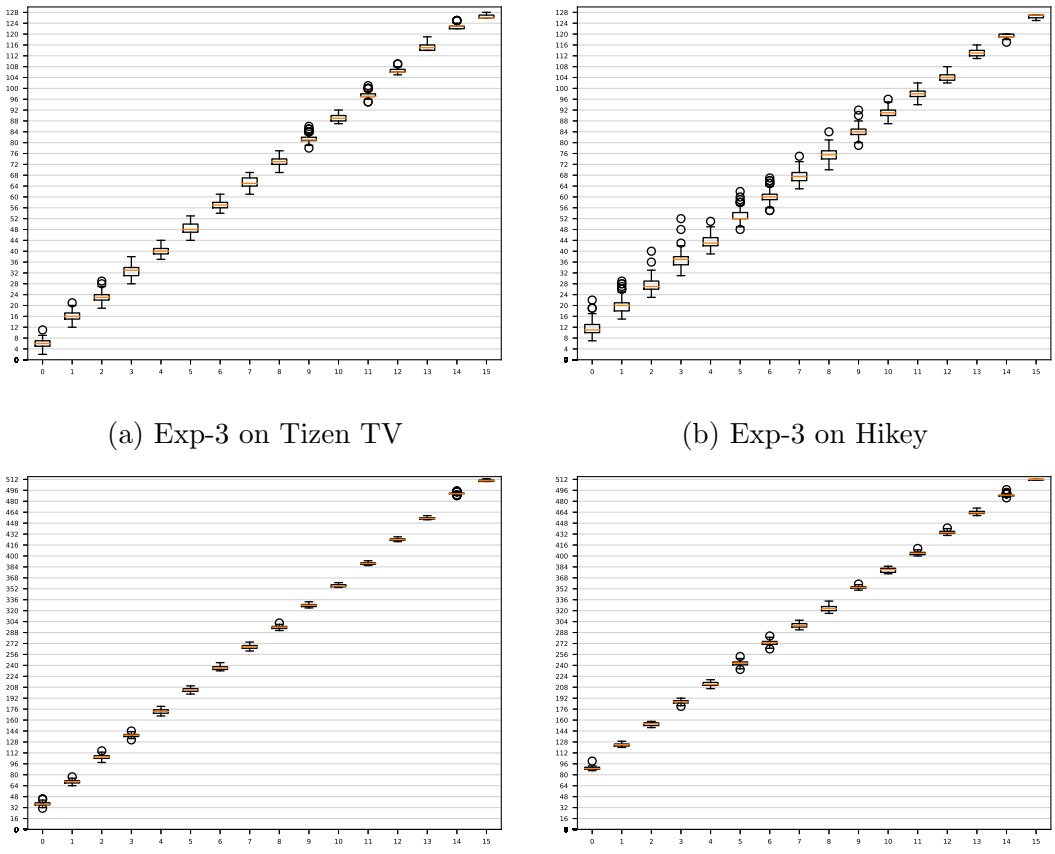
(a) Exp-9 on Tizen TV (b) Exp-9 on Tizen TV

Figure 3.5: Experiment under a Realistic Condition, Where a Youtube Application Is Running.

Under Extreme Conditions

We are interested in how our approach performs under extreme conditions. To this end, we ran a program in the normal world that creates many threads that exceed the number of cores on each board. The threads stay in an infinite loop in which they keep reading and writing data to memory after allocating a memory region that has the same size as the L2 cache.

We conducted multiple experiments with three different configurations: (1) Exp-6: the set-counting mode under the single-core scenario; (2) Exp-7: the line-counting mode under the single-core scenario; (3) Exp-8: the set-v counting mode under the cross-core scenario. Figure 3.4 suggests our approach performs well in the single-core scenario even under extreme conditions. However, the error rate is very high in the cross-core scenario.



(a) Exp-3 on Tizen TV

(b) Exp-3 on Hikey

(c) Exp-4 on Tizen TV

(d) Exp-4 on Hikey

Figure 3.6: The Chosen Bucket Ranges for Different Experiment Configurations Versus Detected Refill Events.

Under a Real-world Condition

In addition, we also tested our approach in the cross-core scenario under a more realistic condition, where a YouTube application was running in the Samsung Tizen TV (Exp-9). As shown in Figure 3.5-(a), the noise was alleviated compared to Figure 3.4-(e) (Exp-8). However, Figure 3.5-(b) implies that the cross-core covert channel is difficult to utilize because there are many overlaps in the ranges of each bucket.

3.6.2 Choosing Bucket Ranges

Figure 3.6 shows the distributions of the number of cache refill events when we select 16 buckets. We assumed that the covert channel is employed to send 4 bits per time. The *Sender* tries to load a specific number of cache lines/sets (x-axis), and the *Receiver* detects how many events are occurred and decodes it to a message (y-axis).

The box and whisker diagram used in Figure 3.6 is to display the distribution of data. Data from the first to the third quartiles is in the box, and the red line inside the box represents the median. The bottom line and the top line represent the minimum and maximum value, respectively. The other small circles are outliers. As shown in Figure 3.6, it is difficult to find overlapped ranges in the line-counting mode after we applied our approaches. On the other hand, in the set-counting mode, the available numbers of the event are smaller than the line-counting mode, and thus, there are overlapping refill event numbers between buckets.

3.6.3 Capacity Measurement

For the capacity measurement, we evaluated how many bytes can be transferred per second using the channels. In particular, we designed 4 experiments: (1) Exp-10: the *Sender* tries to load all cache lines/sets (write all ones to the channel); (2) Exp-11: the *Sender* does not loads anything (write zero to the channel); (3) Exp-12: the *Sender* tries to load all cache lines/sets (write all ones to the channel) under extreme conditions; (4) Exp-13: the *Sender* does not loads anything (write zero to the channel) under extreme conditions; we ran all four experiments 500 times on both devices using different counting modes.

As shown in Table 3.3, the single-core set-counting mode of Exp-11 has the highest capacity and the cross-core of Exp-12 has the lowest capacity for both Hikey board

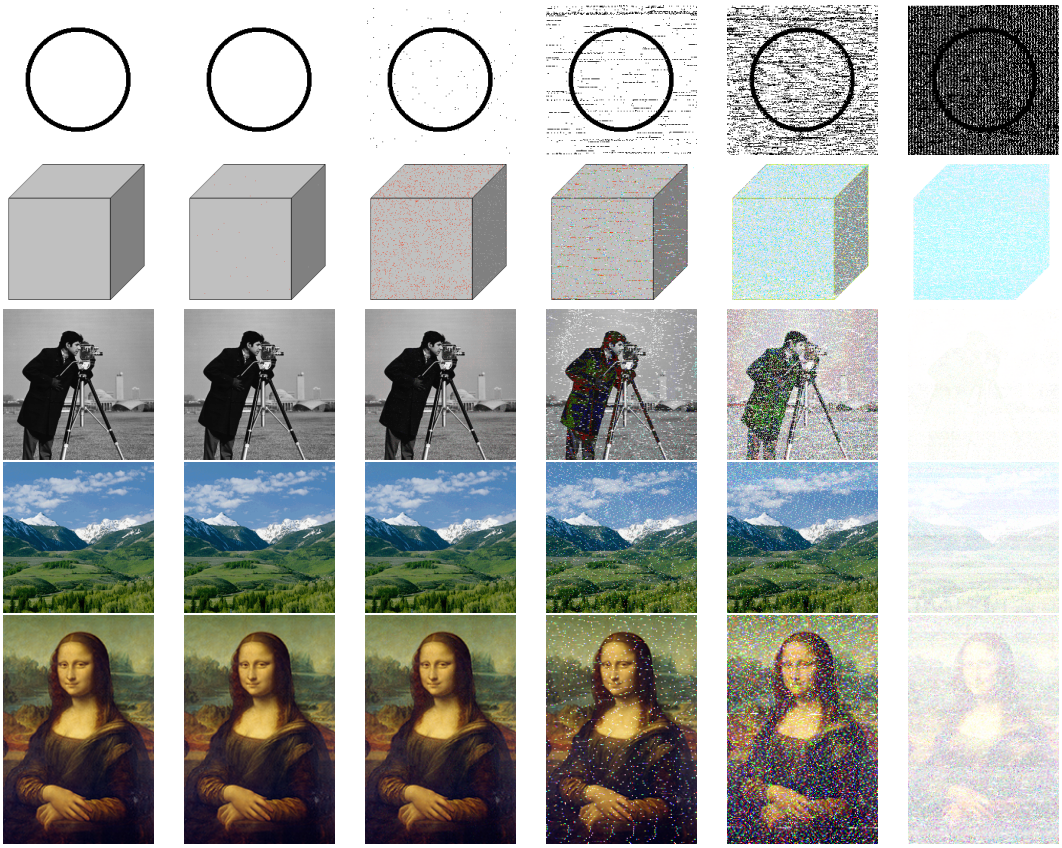


Figure 3.7: Transferred Images Using Covert Channels on Tizen TV: (a) Original images; (b) Single-core line-counting; (c) Single-core set-counting; (d) Cross-core; (e) Cross-core when YouTube is running; (f) Cross-core under extreme conditions.

and Samsung Tizen TV. The results may be surprising at the first glance since our experiments showed line-counting mode has lower noise and 2 more bits to use than set-counting mode. Further analysis reveals the reason behind this phenomenon is that the code of line-counting mode takes much longer time to run than its set-counting mode counterpart. This finding demonstrates the importance of efficient code execution to the covert-channel capacity.

3.6.4 Image Transfer

We used the covert channels to transmit images from the secure world to the normal world under different conditions using both devices. Figure 3.7 shows the results of experiments on the Tizen TV. Column (a) shows the original images. The other images are all the ones we retrieved from the normal world using covert channels.

Overall, the quality and accuracy of the transferred images decrease from left to right; and even under extreme conditions (Figure 3.7 column (f)), the covert channel can still transmit data with some accuracy. The images illustrate that the covert channels we built using PRIME+COUNT are effective.

We especially can transfer data without noise in the single-core scenario using the line-counting mode as shown in Figure 3.7-(b). Because there is no overlapped region between the boxes in Figure 3.6-(c) and (d), we set each bucket to have enough range so that the receiver can decode correct message.

However, the cross-core covert channels have low accuracy particularly when YouTube was running and under extreme conditions as illustrated in Figure 3.7-(e) and (f), Under these conditions where we cannot avoid much noise, the number of cache refill events increases unexpectedly as the Figure 3.4-(e), (f) and Figure 3.5 show. Therefore, message sent by the *Sender* is likely to go other buckets (to higher numbers) because of severe noise.

3.7 Discussion

Limitations of Prime+Count

First off, it is worth noting that covert channels made by PRIME+COUNT could be detected by monitoring PMUs. To detect use of L1/L2 cache refill events, a defender can check the performance monitors event counter selection register (PMSELR) and

the performance monitors selected event type register (PMXEVTYPER) [30].

In addition, PRIME+COUNT is not as fine-grained as other cache attacks, including PRIME+PROBE and FLUSH+FLUSH, because it only cares about how many cache sets/lines have been updated. Adopting PRIME+COUNT for spying a victim program and even extract cryptographic keys from another address space may be difficult if not impossible, because PRIME+COUNT cannot answer which cache sets/lines have been used. However, due to the coarse-grained characteristic of PRIME+COUNT it can reduce noise introduced by world switching, pseudo-random replacement policy, and other factors, which makes it a better choice to build cross-world covert channels.

Cross-world Covert Channels without Normal World Kernel Privileges

To loosen up the attack model and allow normal world applications to use the covert channels, we can adopt the PRIME approach proposed in [91] that can be conducted in userland without using the `DC CISW` instruction. As mentioned in Section 3.3, we can also utilize the Linux *perf_event_open* system call to monitor "L1/L2 cache refill events" in userland to implement COUNT.

Limitations of Our Experiments

While we took great efforts to maintain our experiments' validity, we could not consider some factors that may have affected the bandwidth of the constructed covert channels. Specifically, SeCReT is not openly available (in fact, the authors were unwilling to share their code or system with us), therefore we were unable to run our experiments with SeCReT enabled. It is unclear how much SeCReT or similar solutions would impact the CPU load and even the number of accesses to the cache. We want to emphasize that the deployment of SeCReT or a strong monitor will not affect the feasibility of the proposed covert channels but only downgrade the bandwidth.

Cache Side Channel Attacks

Cache side channel attacks exploit the leakage of information caused by micro architectural time differences between a cache hit and a cache miss [166]. They have been used to steal cryptographic keys in victim programs [113, 161, 151, 152, 95], trace the execution of programs [17, 47, 91], and extract other sensitive information [159, 163, 135, 120]. Even though covert channels can be built using various techniques [49, 125], cache-based covert channel received a lot of attention in recent years [139]. Xu et al. explored cross-VM L2 cache covert channels in Amazon EC2 [146]. Wu et al. designed a high-bandwidth and reliable data transmission cache-based covert channel in the cloud [144]. Maurice et al. characterized noise on cache covert channels and built a robust covert channel based on established techniques from wireless transmission protocols [99].

The Security of TrustZone

SeCReT showed that TrustZone itself cannot guarantee secure communication between normal and secure world [80]. Machiry et al. presented vulnerabilities that permit normal world user-level applications to read and write any memory location in the kernel by tricking secure world into performing the operations on its behalf [98]. ARMageddon demonstrated how to use Prime+Probe to spy code executions on TrustZone [91]. TruSpy demonstrated that it is possible for a normal world attacker to steal a fine-grained secret from the secure world using timing-based cache side-channel [154]. In this work, we presented the first attempt to build cross-world covert channels in the TrustZone architecture.

3.9 Conclusion

In this work, we presented cross-world covert channel attacks on ARM TrustZone, which is designed to provide hardware-assisted isolation. We demonstrated that existing channel protection solutions, such as SeCReT, or even more powerful mechanisms, such as a strong monitor, can be bypassed. We discussed the reasons why previous attacks, including Prime+Probe and Flush+Reload, do not work for the cross-world scenario on ARM. And, we designed a low noise, no shared memory needed cache attack named Prime+Count by leveraging overlooked PMU "L1/L2 cache refill events." Our experiments showed that Prime+Count-based cross-world covert channels could achieve bandwidth as high as 27 KB/s under the single-core scenario and 95 B/s under the cross-core scenario.

Chapter 4

SmokeBomb: EFFECTIVE MITIGATION AGAINST CACHE SIDE-CHANNEL ATTACKS ON THE ARM ARCHITECTURE

4.1 Introduction

Cache side-channel attacks exploit time differences between a cache hit and a cache miss to infer sensitive data [113]. These attacks are very effective in stealing cryptographic keys (*e.g.*, the secret keys used in AES) from victim programs and even other virtual machines, in tracing the execution of programs, and in performing other malicious actions [95, 151, 152, 162, 17, 47, 91, 120, 136, 160, 164].

The bulk of research into cache attacks was started on, and has continued on, the Intel architecture. However, as mobile devices (*e.g.*, smartphones and watches) experience unprecedented growth, the mitigation of cache attacks on non-Intel architectures has drastically risen in importance. Thus, our community needs a defense against cache side-channel attacks that can protect against both shared cache (L2) attacks and dedicated cache (L1) attacks, and can do so regardless of the specific architecture in question. While cache attacks are, generally, more difficult to carry out on the ARM architecture, new techniques have been developed to make them more effective on mobile platforms [66, 91, 158, 155].

The fundamental causes of cache side-channel attacks are two-fold: (1) a cache is a hardware resource shared by multiple processes; and (2) there is a noticeable difference in access time between a cache hit and a cache miss. Therefore, to fundamentally solve this problem, a new architecture or cache design is needed. Such hardware-based approaches can provide strong security features against cache attacks with relatively

small performance overhead [85, 82, 60, 140, 94]. Hardware-based solutions, however, require considerable cost and time to be deployed in a practical manner, *and no such concerted effort has yet been undertaken.* Thus, current systems remain vulnerable to current cache attacks, and even if a hardware solution is undertaken, legacy devices will not be secured.

However, software-based approaches are relatively cheap and easily deployable: We can deploy them quickly and broadly through software patches. Therefore, understandably, many software solutions have been proposed to mitigate cache attacks [69, 84, 93, 165, 167, 78]. Some techniques target the protection of the *shared CPU cache* (*i.e.*, L2 cache in the ARM architecture), meaning that they fail to protect programs from emergent attacks against the *dedicated core cache* (*i.e.*, L1 cache) [84, 167, 93, 165, 91, 78, 155, 66]. Crane et al. [55] reduce, but does not eliminate, side-channel information leakage by randomizing the program's control flow. Other techniques, including recent work targeting the protection of L1 cache, use specific hardware features available only on *certain Intel processors* and have uncertain efficacy under heavy system load [69]. In analyzing these techniques, we realized that most current cache side-channel protection mechanisms attempt to mitigate attacks by implicitly creating a *private space*—not shared with any other process—in which constant-time (and thus, side-channel-immune) access to sensitive data is assured.

In this work, we propose SmokeBomb, a software cache side-channel mitigation method for commonly-used CPU cache configuration, that *explicitly* ensures a private space for each process to safely access critical data—the actual L1 cache of the CPU core on which the process is executing. SmokeBomb reserves the L1 cache for a sensitive operation's exclusive use and *denies attackers the ability to find timing differences between used and unused sensitive data.* Without access to measurable time differences, attackers are unable to carry out cache-based side-channel attacks.

SmokeBomb employs additional OS-level functionality (uninvasively implemented as a kernel module), which has zero performance impact when there is no sensitive data to protect, and negligible impact on the rest of the system. While SmokeBomb requires a recompilation of the sensitive code that needs to be protected, it assists developers in adopting the protection as a compiler extension, requiring developers to *only annotate the sensitive data*.

We demonstrate the effectiveness of SmokeBomb by applying it to two different ARM processors with different instruction set versions and cache models and carry out an in-depth evaluation of the efficiency of the protection. Our experimental results show that SmokeBomb effectively prevents information leakage against known cache side-channel attacks. To our knowledge, SmokeBomb is the first cache side-channel defense that functions on the ARM architecture and covers both the L1 and L2 cache layers.

4.2 Cache Side-channel Attack Methods

Evict+Time

This attack method can determine which cache sets have been used by the victim process [91, 65]. In the first step, the attacker measures the execution time of the victim process. Then, the attacker evicts a target cache set and measures the execution time of the victim program again. From the difference in the execution time, the attacker can figure out the cache set and, thus, the memory that it represents has been used by the victim program.

Prime+Probe

This attack is also used to determine specific cache sets accessed by the victim. It has been studied and implemented in various environments [162, 77, 95, 74, 155, 71,

91, 46, 122, 73].

In the Prime phase, the attacker occupies a certain range of cache sets by loading their own data. After the victim process has been scheduled, the attacker probes which cache sets are used by the victim. Because the ARM architecture uses a set-associative cache, a set consists of several *ways*. For example, the L2 cache of the Cortex-A53 has 16 *ways*. Thus, the attacker decides a set has been used if one of the *ways* was refilled by other data, which might not be loaded by the victim. The pseudo-random cache replacement policy of the ARM architecture makes the Prime+Probe attack much more difficult [91].

Flush+Reload and Evict+Reload

These attacks operate by measuring the data reload time of the cache, which are available only when the attacker shares memory with the victim process [72, 78, 152, 158]. The attacker must map a target shared object into its address space. Then, the attacker flushes/evicts a cache line within the shared area. In the Reload phase, attackers reload previously flushed/evicted data after waiting for the victim to access the shared object and checks the time it takes to reload. Based on this reloading time, attackers can infer if the victim accessed the data.

Commonly, attacks can check whether specific data is in the cache after the execution of the victim process. These attacks are more accurate and easier to conduct than the Prime+Probe and Evict+Time attacks. Also, the simplicity of these methods makes asynchronous attacks possible. The only difference is in the way for flushing the data from the cache before the victim has been scheduled. If the flush instruction is available, the attacker could flush data using the virtual address. Otherwise, the attacker needs to evict the data by loading other data [70, 91].

Flush+Flush

This attack utilizes the timing difference of the flush instruction [68, 91]. The execution time of the flush instruction is different depending on whether data is cached or not. If data is cached, the flush instruction takes more time to execute. The first phase is identical to the FLUSH+RELOAD attack. In the last step, however, the attacker flushes the data once again to check whether the data has been accessed by the victim.

4.3 Our Threat Model

In this work, we consider multi-core computing environments that use the inclusive and the non-inclusive caches on the ARM architecture, in which processes, including malicious ones, use shared libraries, such as OpenSSL. The attacker can use all of the cache side-channel attacks mentioned in Section 4.2 to extract secret information including cryptographic keys. We do not consider recently proposed PRIME+ABORT, because the ARM architecture currently does not support transactional memory, and therefore the attack is not available [59]. In addition, we assume the worst case scenario in which attackers can use the flush instruction on ARMv8 CPUs, even though the ARMv8 architecture restricts userland applications from executing the flush instruction by default. Assuming this worst-case attack model allows us to develop as strong a defense as possible, which we present in the rest of the work.

Throughout this work, we discuss and perform experiments on two environments as listed in Table 4.1.

4.4 Overview

We present, implement, and evaluate SMOKEBOMB—a cache side-channel mitigation method that prevents attacks on every cache level mentioned in Section 2.2,

Table 4.1: Test Environments.

CPU (# of cores)	Instruction Set	L1-D Cache	L2 Cache	Inclusiveness	Cache Replacement Policy
Cortex-A72 (4)	ARMv7 32-bit	32 KB, 2-way, 256 sets	2 MB, 16-way, 2048 sets	Inclusive	Least-recently-used
Cortex-A53 (4)	ARMv8 64-bit	32 KB, 4-way, 128 sets	512 KB, 16-way, 512 sets	Non-inclusive	Psuedo-random

especially for inclusive and non-inclusive caches. With SmokeBomb, an attacker attempting to measure data access times of sensitive data will be met with consistent timing results for all sensitive data, and *will thus be unable to infer which sensitive data is actually used*. To this end, we design SmokeBomb to achieve particular defensive goals:

(**D1**) It defends against cross-core L2 data cache attacks.

(**D2**) It defends against directory protocol based cross-core L1 data cache attacks.

(**D3**) It defends against single-core L1 data cache attacks.

To accomplish these goals, SmokeBomb first instruments applications (during compilation time) to find and patch any sensitive code (Section 4.5), which puts the sensitive data involved under SmokeBomb's protection (where the developer annotates the sensitive data in the source code). Then, SmokeBomb carries out three steps, at different points before, during, and after the execution of the patched sensitive code. Figure 4.1 depicts the effect of SmokeBomb in terms of the amount of data loaded in the cache, and we reference it through the following description.

Preloading Sensitive Data (Section 4.6): Before executing the sensitive code (at time-point t_1 in Figure 4.1), SmokeBomb preloads the sensitive data into the L1 cache. By preloading the sensitive data, SmokeBomb prevents the attacker from identifying which specific cache *sets* have been used by the victim (*e.g.*, Prime+Probe).

Preserving Sensitive Data (Section 4.7): SmokeBomb ensures that the preloaded data exists in the private L1 cache *throughout the sensitive code's execution* ($t_1 \sim t_2$ in Figure 4.1). In a *non-inclusive* cache, the preloaded data will be maintained only in the L1 cache. In an *inclusive* cache, the preloaded data will be maintained in the L1 and the L2 cache. Thus, an *asynchronous* attacker from another core cannot infer any information regarding the key data.

Flushing Sensitive Data (Section 4.8): At the termination of the sensitive code (t_2 in Figure 4.1), SmokeBomb flushes all data from the cache so that no information on what data was used is revealed to an attacker (between time points t_2 and t_3).

Without SmokeBomb, as the sensitive code executes from t_1 to t_2, the amount of key data in the cache increases gradually. At t_2, all the key data may have been fetched into the cache and will stay there until being replaced gradually as shown from t_2 to t_3. Thus, attackers can infer which sensitive data is the key data from t_1 to t_3. With SmokeBomb, however, entire or only a certain amount of sensitive data is fetched into the cache when sensitive code starts execution at t_1 and flushed when sensitive code exits at t_2. Consequently, SmokeBomb will cause consistent timing results for all sensitive data. We note that SmokeBomb is not able to defend instruction cache attacks, which will be discussed in Section 5.7.

Instructions Used

SmokeBomb uses the data prefetcher by calling preloading data (`PLD`) instruction to preload the sensitive data. Even though ARM architecture reference manuals state that "the effect of an the `PLD` instructions is *implementation defined*", we have confirmed that *all* the Cortex-A processors support `PLD` instructions and their effects

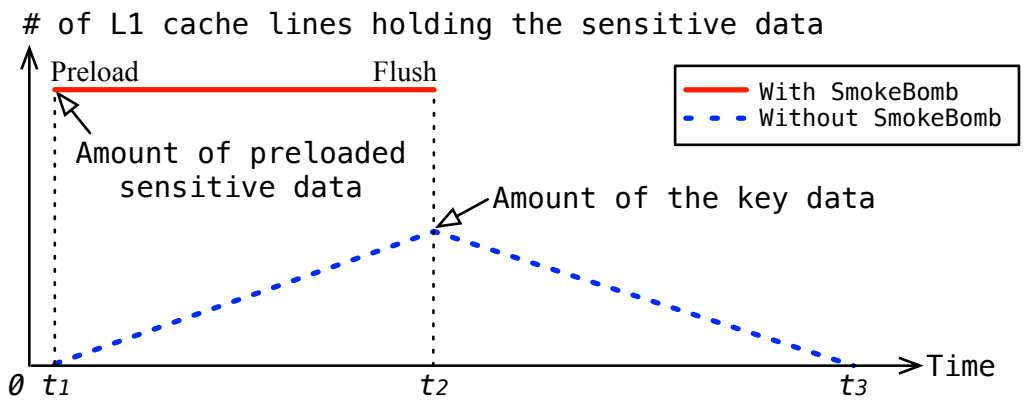


Figure 4.1: The Difference in Cache Usage with and without SmokeBomb: The x-axis denotes the time of code execution and the y-axis represents the number of cache lines holding sensitive data. Any observable changes on the y access represent a potential cache side-channel attack vector.

through the technical reference manuals of each Cortex-A processor [32, 26, 21, 24, 25, 28, 40, 41, 33, 42, 34, 35, 36, 43, 27, 31].

The `DC CISW` instruction flushes a designated cache line in a specific cache level [27, 31]. It requires a set number, a *way* number, and a cache level as operands. SmokeBomb uses the `DC CISW` instruction to bypass the psuedo-random replacement policy. Also, this instruction is used to keep sensitive data away from the L2 cache in inclusive caches.

SmokeBomb APIs

In applications, SmokeBomb is initiated and finalized by the following two APIs in user-space programs: (1) `init_smokeBomb`, and (2) `exit_smokeBomb`. The `init` API has two parameters: the start address and the size of the sensitive data. When this API is called by a process in the system, SmokeBomb preloads the sensitive data into the cache and changes the scheduling policy of the process. The `exit` API, which

flushes the sensitive data from the cache and restores the scheduling policy, does not have any parameters. Between these two APIs—during the execution of the sensitive code, SmokeBomb-defined instructions execute to preserve the preloaded sensitive data in the L1 cache.

SmokeBomb-defined Instructions

Because most ARM processors have no cache locking instructions, we utilize the undefined instruction exception handler to implement our own cache locking instructions that are software-emulated by the handler. SmokeBomb finds and patches cache-relevant instructions (such as `LDR` or `STR`) that access *non-sensitive* data, because they can change the cache state by fetching *non-sensitive* data to the L1 cache. At runtime, those instructions will be trapped and handled by SmokeBomb's exception handler. we call the patched instructions as `xSB` instructions, such as `LDRSB`, which performs the intended operation of the original `x` instruction, but also ensures the preservation of the sensitive data only in the L1 cache.

4.5 Instrumenting Sensitive Code

SmokeBomb requires two modifications to the sensitive code. First, the two API calls mentioned in Section 6.5 must be inserted before and after the sensitive code. Second, cache changing instructions in the sensitive code must be modified to SmokeBomb-defined instructions which have opcodes that do not exist in the ARMv7 and ARMv8 instruction sets. SmokeBomb software-emulates them through the undefined instruction handler.

SmokeBomb automates this process for developers by requiring only an annotation of the sensitive data (in our implementation, using attribute syntax annotations [13]), and SmokeBomb derives all necessary code modifications during compi-

lation. Developers can annotate static data directly or a data pointer for dynamically allocated data. Note that, as SmokeBomb is a *protection* mechanism, we rely on the developers to identify the data that *should* be protected. However, approaches exist for the automated identification of such data, and this compilation process could be modified to automatically insert even the annotations themselves [137]

Provided these annotations, SmokeBomb uses a compiler extension (*i.e.*, an LLVM pass) to instrument the application. First, SmokeBomb identifies the sensitive code, which is straightforward due to the annotation. By analyzing each IR instruction, SmokeBomb can identify all memory operations that reference (annotated) sensitive data. Specifically, when data is annotated, SmokeBomb can identify all memory operations that reference annotated sensitive data by analyzing operands of load and store instructions. When a pointer is annotated, SmokeBomb checks whether memory operations dereference the annotated pointer or not. All such instructions are identified as *sensitive code*.

Once sensitive code is found in a function, SmokeBomb identifies the dominator and post dominator of the basic blocks in which the sensitive code exists. It then inserts a call instruction which invokes the `init` API at the dominator node (and specifies the reference to the sensitive data in the API call) and another call instruction for the `exit` API at the post dominates node. If there are other call instructions in the basic blocks, SmokeBomb additionally places instructions calling the `exit` API and the `init` API before and after the other call instructions, respectively.

When the size of the sensitive data is larger than the L1 cache, SmokeBomb takes the first part of the sensitive data as *selected* sensitive data that will be preloaded and preserved. The selected sensitive data is bigger than *a way* of the L1 cache so that the selected part can cover all *sets* for preventing attacks which try to identify which set is used (*e.g.*, Prime+Probe). Note that *unselected* sensitive data is neither preloaded

nor preserved: it is explicitly kept *out* of the cache, achieving the same protection.

Next, SmokeBomb patches all cache changing instructions that are located between the two APIs. When the size of sensitive data is smaller than the L1 data cache, it only patches instructions that access *non-sensitive* data, to preserve sensitive data in the L1 cache by enforcing additional cache maintenance operations after these instructions execute. The insight here is that instructions that access *sensitive* data do not change the cache state of that data. By patching only *non-sensitive* instructions, SmokeBomb can avoid unnecessary undefined instruction exceptions, thus minimizing performance degradation. However, if the size of sensitive data is larger than the L1 data cache, SmokeBomb patches *all* cache changing instructions, because it cannot statically determine which sensitive code might access the *unselected* sensitive data.

4.6 Preloading Sensitive Data

Before sensitive code executes, SmokeBomb preloads the sensitive data (or, if the sensitive data larger than the L1 cache, the selected sensitive data). One way to preload data into the cache is to simply access it (e.g., using the `LDR` instruction). However, this is slow, as the CPU will wait until the data actually arrives in a register or memory. For better performance, SmokeBomb employs a hardware feature called *data prefetching*, by using the preloading data (PLD) instruction, which is available in the Cortex-A series. SmokeBomb triggers the prefetcher by using the `PLD` instruction in ARMv7 and the `PRFM PLD` instruction in ARMv8 [27, 31]. For brevity, we use "`PLD` instructions" to refer to both instruction forms. `PLD` instructions execute much faster than `LDR` to fetch data into the cache.

Bypassing the Pseudo-random Replacement Policy.

The `PLD` instruction loads data (of the size of a cache line) from memory to the cache. However, with ARM's pseudo-random cache replacement policy, sensitive data loaded earlier in the process might be evicted by sensitive data loaded later in the process. SmokeBomb must ensure that this does not happen, so that the entire sensitive data can be safely loaded. Our experiments on both testing environments reveal that the pseudo-random replacement policy only triggers when there is no empty cache line available. If we can make one cache line available by flushing it in the set that the data is supposed to reside, the data is guaranteed to occupy the empty cache line instead of evicting any other line in the set.

We conducted preliminary experiments to confirm this behavior for both the L1 and L2 caches: We first flush a particular cache line of a set using the `DC CISW` instruction, which takes a set number and *a way* number as operands [27, 31]. Then, we load data using the `PLD` instruction from an address whose index fields match the set number. We then flush the same cache line again. At the last step, we load the same data again on the same core and check the cache refill event using the performance monitor unit (PMU) [27, 31]. If a cache refill event occurs, the data has been loaded in the cache line we selected, and vice versa.

Keeping Sensitive Data Away from L2 Cache.

For non-inclusive caches, `PLD` instructions load data into L2 cache automatically as well, which enables cross-core cache attacks because an L2 cache line is evictable. To prevent this, we use the same approach described in the previous subsection to ensure that sensitive data is always loaded into a known *way* in the cache. Then, we flush the sensitive data from the L2 cache. When SmokeBomb preloads the

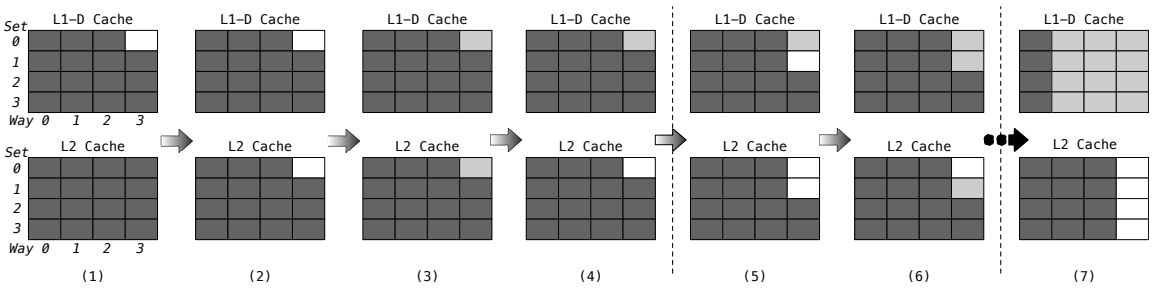


Figure 4.2: Example of the Cache State Changes in the Non-inclusive Cache Model When Preloading the Sensitive Data: In this figure, we assume that there are 4 sets and 4 *ways* in both the L1 and L2 caches. Light gray means *sensitive data* and dark gray means *normal data*. White represents a *flushed cache line*. (1): Flush an L1 cache line. (2): Flush an L2 cache line of the last *way*. (3): Load the sensitive data. (4): Flush the sensitive data from the L2 cache. (5)–(7): Repeat the prior 4 steps.

sensitive data, other processes' data in the L2 cache can be evicted. To minimize this impact, SmokeBomb uses only the last *way* of the L2 cache. For inclusive caches, SmokeBomb loads the sensitive data into the L2 cache.

Figure 4.2 illustrates how SmokeBomb preloads the sensitive data into the cache, bypassing the pseudo-random replacement policy. We first translate the virtual address of the sensitive data to physical address and compute its set number for L1 and L2 cache respectively, because the cache is physically indexed and tagged. We then flush the one way of this set in L1 and L2 cache respectively to make room for sensitive data as shown in (1) – (2) of Figure 4.2. For convenience, we flush from the last *way* of L1 to the first *way* in this step. Then, we use `PLD` instructions to load data into the cache as shown in (3) of Figure 4.2. Because there is one cache line available in L1 and L2 cache, the data goes to that available line. We flush the just loaded L2 cache line as shown in (4) of Figure 4.2. We repeat this procedure until the entire sensitive data or the selected sensitive data is loaded into the cache. In

this loop, if a *way* of the L1 cache is fully occupied with sensitive data, we start to fill the previous *way* instead as shown in (5)–(7) of Figure 4.2. For inclusive caches, we can omit to flush an L2 cache line—step (4) of Figure 4.2. SmokeBomb changes a *way* of the L2 cache similar to the L1 from the last *way* in descending order, if a *way* is full with sensitive data.

4.7 Preserving Sensitive Data

After preloading the sensitive data (or the selected sensitive data, if the sensitive data is larger than the size of the L1 cache), it is critical to preserve it in the cache during the execution of sensitive code to prevent side-channel attacks. Additionally, unselected sensitive data must *not* be in the cache. By preserving only preloaded sensitive data, SmokeBomb achieves a consistent cache state throughout the execution of the sensitive code.

In our experimentation environment, we tested how many cache lines were evicted during AES encryption after preloading the T-tables as the sensitive data into the L1 cache. We used OpenSSL (v.1.0.2) and 128-bit AES algorithm to encrypt an 8-byte plaintext. Even with an intentionally-chosen small plaintext, the results show that around 89% of cache lines holding non-key sensitive data were evicted during the encryption procedure, which opens the door for cache side-channel attacks. The experiments clearly demonstrate the need to preserve sensitive data in the cache.

Unfortunately, most ARM Cortex-A series processors do not support hardware cache locking techniques. Thus, to preserve the sensitive data in the L1 cache, SmokeBomb must hook all instructions that could influence the state of the cache after preloading the sensitive data. If the sensitive data is smaller than the L1 cache, SmokeBomb only needs to hook cache changing instructions that access *non-sensitive* data, because these instructions can evict the preloaded sensitive data from the

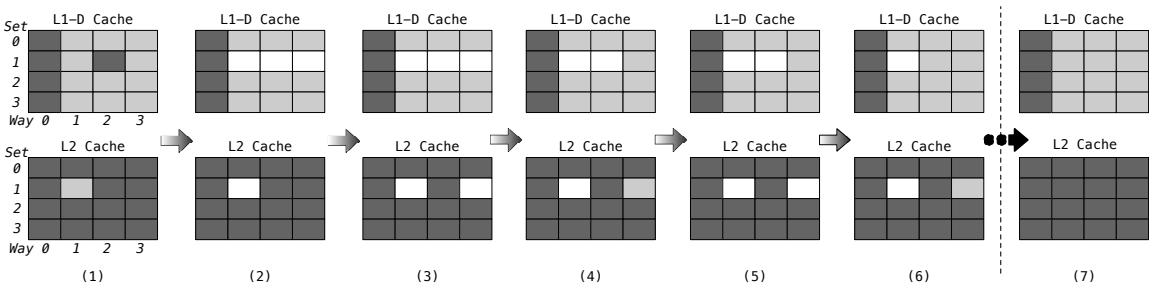


Figure 4.3: Example of the Cache State Changes in the Non-inclusive Cache Model When Preserving the Sensitive Data: In this figure, we assume that there are 4 sets and 4 *ways* in both the L1 and L2 caches. Light gray means *the sensitive data* and dark gray means *normal data*. White represents a *flushed cache line*. (1): Data, which is not in the L1 cache, is loaded. (2): Flush the sensitive data located in the set of the L1 cache. (3) – (7): Refill the set with the sensitive data again.

L1 cache to the L2 cache. When the sensitive data is larger than the L1 cache, SmokeBomb must hook all cache instructions that occur in sensitive code regardless of which data they access.

To design a software cache locking technique, we use the undefined instruction exception handler, which can be used to implement custom "soft-instructions."

Handling xSB Instructions.

SmokeBomb installs an undefined instruction handler for each `xSB` instruction. We use a handler for `LDRSB` to illustrate how the handlers work with the non-inclusive cache model (as shown in Figure 4.3).

SmokeBomb first loads data referenced by the original instruction. If the address of this data is already in the L1 cache, no matter if the data is sensitive or not, the handler returns immediately to the sensitive code. This is because if *non-sensitive* data is already in the cache after preloading the sensitive data, it means the data exists

in the cache with the sensitive data. We determine if the data is in the L1 cache by checking the L1 data cache refill event. If the event did not occur (*i.e.*, an L1 cache hit occurred), the memory system does not fetch the data from the main memory. Consequently, the preloaded sensitive data is still only in the L1 cache.

If the data is neither non-sensitive data that is already in the L1 cache nor preloaded sensitive data, the data will be fetched into the L1 and L2 caches as shown in (1) of Figure 4.3, which may result in the eviction of a cache line where the sensitive data or selected sensitive is stored. Because we cannot determine which *way* has been evicted due to the pseudo-random replacement policy, SmokeBomb simply reloads the sensitive data *in the set* as in the preload procedure. However, if the address of the loaded data is not congruent with any preloaded sensitive data, SmokeBomb returns to the sensitive code without reloading.

To reload the sensitive data, SmokeBomb flushes it located in the *set* using its virtual address. Once the sensitive data has been evicted from the L1 cache, SmokeBomb cannot know which *way* of the L2 cache has the data. Therefore, SmokeBomb entirely removes the sensitive data in the *set* from a cache as shown in (2) of Figure 4.3. Then, SmokeBomb reloads the sensitive data following the preloading method to fill the *set* again as shown in (3) – (7) of Figure 4.3. As a consequence, the process has the same sensitive data that was preloaded only in the L1 cache.

In the inclusive cache, everything is identical except that we omit flushing an L2 cache line when reloading the sensitive data.

Handling Preemption.

Modern operating systems have a preemptive kernel and provide preemptive multitasking features. These systems allow scheduled processes to execute only for a

time slice. For example, the Completely Fair Scheduler (CFS), which is the default scheduler of the Linux kernel, interrupts a process when the time slice of its thread is expired. Context switches can also occur when a thread voluntarily yields control of the CPU by making system calls, such as `sleep` and `yield`.

Unfortunately, a context switch can cause an attacker process to be executed by the core that was previously running the sensitive code, allowing it to influence the state of the L1 cache. To avoid potential attacks stemming from this phenomena, SmokeBomb must do one of two things during context switches: (1) it must flush (on preemption of sensitive code) and re-preload (on resumption of sensitive code) the sensitive data or (2) it must prevent preemption from happening in the first place. The latter represents minimal change to the running system itself.

SmokeBomb overcomes this challenge by executing the sensitive code of a process on the same core until it returns (from `init_smokeBomb` to `exit_smokeBomb`). This *guarantees* that the L1 cache subordinated to the core is not being used by any other processes while executing the sensitive code between the two APIs.

To prevent preemption, SmokeBomb temporarily changes the scheduling policy of only *the single process* when it starts to execute the sensitive code (other processes on the system continue running under the default scheduling policy). Unlike kernel threads, which can manipulate the preemption strategy itself, user-level threads cannot be free from preemption. However, a user-level process can run until it relinquishes the CPU voluntarily by using the First-In, First-Out (FIFO) scheduling policy with the highest static priority. Among the available scheduling policies in the Linux kernel, the FIFO scheduling is the only policy that will not schedule a thread in the time slice manner [16].

It is worth noting that SmokeBomb is only activated in a function where sensitive code exists. If other functions are called from the function in which SmokeBomb is

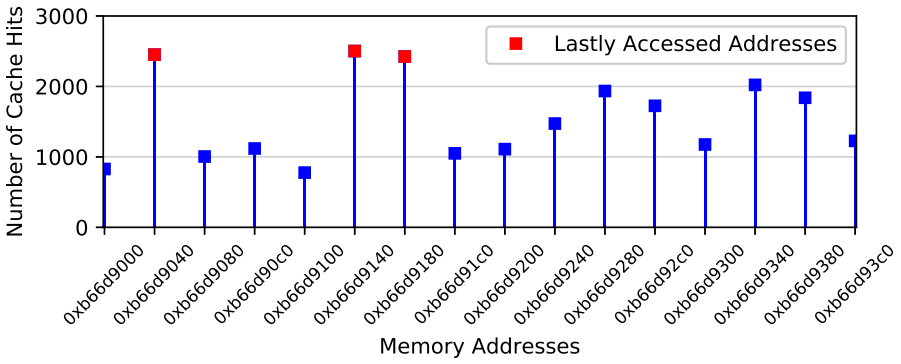


Figure 4.4: The Cache Attack Results of Exploiting the LRU Cache Replacement Policy on the Cortex-A72: The results show that the remaining sensitive data in a cache can be resulted in the key data leakage.

started, `exit_smokebomb` call will be invoked, restoring the scheduler to the default setting. When a thread returns to the function, SmokeBomb is activated again. While this design choice increases the latency on the single application that executes sensitive code, it minimizes the performance impact on the rest of the system.

4.8 Flushing Sensitive Data

After the sensitive code finishes or when the protected process is scheduled out, the entire sensitive data will be gradually evicted if SmokeBomb does not flush it. Even though it seems harmless to leave sensitive data in the cache after the sensitive code terminates, we designed experiments to verify the necessity of flushing: in particular, whether cache exploitation by an attacker is possible when the LRU replacement policy is used. Usually Cortex-A series use the pseudo-random replacement policy, but the LRU policy can be chosen alternatively, and ARM Cortex-A57 and A72 processors employ the LRU replacement policy for the L1 cache by default [25, 40, 41, 35, 34, 35].

We conducted experiments on the Cortex-A72, which uses the LRU replacement policy for the L1 cache. The victim then terminates itself right after accessing three

different memory addresses. Then, the attacker loads data that is congruent with the sensitive data to evict recently used cache lines. At the last step, the attacker checks access times of the sensitive data.

We ran this experiment 3,000 times. Figure 4.4 shows the attack results where the three addresses that the victim actually accessed have the largest number of cache hits (*red squares*) among the preloaded sensitive data. *Blue squares* stand for the preloaded memory addresses that the victim did not access, which have lower number of cache hits than *red squares*. To protect against this attack, SmokeBomb flushes the sensitive data from cache upon termination of sensitive code to prevent information leakage.

4.9 Evaluation

Our experimental environments consist of a Samsung Tizen device and Raspberry PI3 using the Cortex-A72 and A53 processor models, respectively, as listed in Table 4.1. These devices have different instruction sets: ARMv7 and ARMv8. We implemented proof-of-concept prototypes of SmokeBomb for both instruction sets. The prototypes consist of two parts: (1) an LLVM pass with a binary patching tool and (2) a loadable kernel module. SmokeBomb, which can be deployed without requiring changes to the operating system beyond loading the kernel modules and can be adopted by developers by annotating sensitive data in their applications (or using an approach for automatic identification of it, such as CacheD [137]).

Of the discussed attacks, we evaluate against Flush+Reload and Evict+ Reload for the Cortex-A53 (non-inclusive cache) and the Cortex-A72 (inclusive cache), respectively, because these are the fastest and most accurate attack methods. We also used Prime+Pro- be. Except when Prime+Probe is used, all experiments were conducted on a *cross-core* environment, using two processes: the

attacker and the victim. By utilizing a multi-core environment, SmokeBomb's defense against the directory protocol of the ARM architecture is evaluated as well [78]. Flush+Flush was not used for the evaluation because the effectiveness of the method and results are very similar to Flush+Reload and Evict+Reload. Also Evict+Time was not evaluated, because SmokeBomb always loads and flushes the same amount of sensitive data into the cache and the sensitive data does not exist in the cache when SmokeBomb is inactivated, which implies the attack is unavailable.

For the Cortex-A53 processor, which uses the non-inclusive cache, we assume the flush instruction is unlocked for user-land applications, *which strengthens the attacker and makes* SmokeBomb*'s job more difficult*. The Cortex-A72 processor employs the inclusive cache, known as AutoLock, and thus data cannot be evicted by other processes while it is in the L1 data cache [66]. Furthermore, the flush instruction is *not available* in user-land, because the device uses the ARMv7 instruction set. To conduct cache side-channel attacks on the Cortex-A72 processor, we use the two different assumptions introduced by Green et al., depending on the attack method [66]. The victim process provides several services to other processes, and the attacker can request the services. This assumption makes the victim perform the L1 data cache line evictions itself by requests from the attacker. Eventually, the sensitive data of the targeted service can be evicted by the other services requested by the attacker so that Evict+Reload is possible. For Prime+Probe on the Cortex-A72, we assume the attacker and the victim run on the same core. This assumption is theoretically possible, because preemption is disabled only when the sensitive code is executing.

Throughout this section, we present a number of figures describing the difference in cache measurement opportunities for attackers with and without SmokeBomb. In these figures, attack results are shown by *blue squares* in all figures for the non-SmokeBomb case, and by *red circles* for the results of SmokeBomb's application.

4.9.1 *Effectiveness of L1 Cache as a Private Space*

By using the cache refill event of the PMU, we tested if the approaches proposed in Section 4.6 and Section 4.7 to preload sensitive data to cache and preserve it in L1 cache alone works in the non-inclusive cache model using the following two experiments [27, 31].

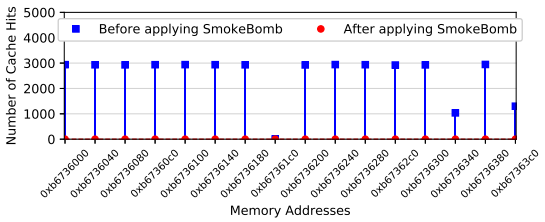
(1) We first loaded 8 KB of data using the `PLD` instruction. Second, we flushed L2 cache using the `DC CISW` instruction. Third, we loaded the same data again with the `LDR` instruction, checking the L2 cache refill event. As expected, no L2 cache refill event occurred because all data accesses triggered L1 cache hit.

(2) In the second experiment, the first and the second steps are as same as in the first experiment. The third step was done by *a different core* and used the `PRFM PLDL2KEEP` instruction for loading the data into L2 cache. This instruction does not fetch data to L1 cache but only to L2 cache for data preloading. The L2 cache refill event occurred, which confirmed that the data was successfully flushed in step 2. If the L2 cache had the data, the event counter would not increase. These experiments clearly demonstrate that SmokeBomb can keep sensitive data in L1 data cache alone.

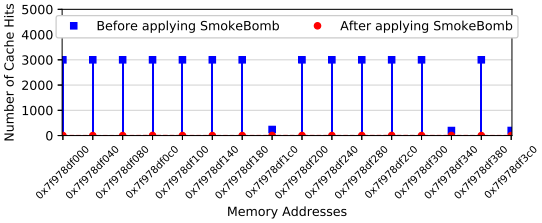
4.9.2 *Security Analysis*

Non-inclusive Caches

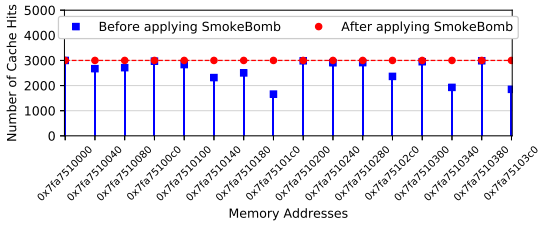
SmokeBomb achieves each defensive goal described in Section 6.5: **D1**—by keeping the sensitive data away from L2 cache, the attacker cannot observe any sensitive data in L2 cache. If sensitive data exists in L2 cache, the other processes can evict the sensitive data, which in turn results in key data leakage; **D2**—by preloading the sensitive data and keeping the sensitive data during sensitive code execution, the



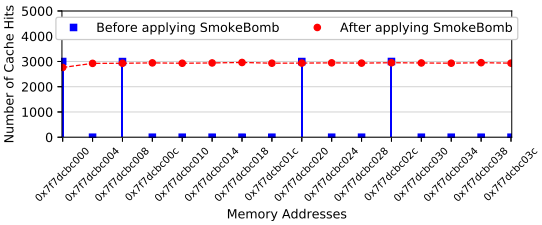
(a) Last-round Attack on Cortex-A72.



(b) Last-round Attack on Cortex-A53.



(c) Last-round Attack on Cortex-A72.



(d) One-round Attack on Cortex-A53.

Figure 4.5: The Attack and Protection Results on the AES Algorithm: In (a), (b), and (d), the Evict+Reload was used on the Cortex-A72 and the Flush+Reload was used on the Cortex-A53. (c) is the results when the Prime+Probe was used on the Cortex-A72

attacker cannot find the key data using the directory protocol; **D3**—by flushing the sensitive data and protecting the sensitive code from preemption.

Inclusive Caches

D2 and **D3** are guaranteed in the non-inclusive cache model. However, SmokeBomb does not need to drive the sensitive data out from L2 cache to achieve **D1**. In inclusive caches, the sensitive data cannot be evicted from L2 cache as long as the sensitive data is in L1 cache. Therefore, SmokeBomb can achieve **D1** for the inclusive cache model by keeping the sensitive data in a cache until the sensitive code execution finishes.

The size of protected data

SmokeBomb can provide the same level of defense even when the size of the sensitive data is larger than L1 data cache. If SmokeBomb detects large sensitive data at compile time, it marks a subset of the sensitive data as *selected* sensitive data, and this data is preloaded and preserved. Next, while SmokeBomb is activated at runtime, it flushes all *unselected* sensitive data *out* of L1 cache during the same operation that maintains the selected sensitive data *in* L1 cache. The *unselected* sensitive data, thus, cannot remain in the cache. Consequently, SmokeBomb can always produce consistent results against cache side-channel attacks by preserving the *selected* sensitive data only. However, with caching essentially disabled for the unselected sensitive data, operations on this data will understandably be slow.

Against asynchronous Flush+Reload/Flush attacks

Sensitive data leakage might occur even with the use of SmokeBomb, which must meet the following conditions: (1) immediately after the preloading phase, an attacker can flush the sensitive data on another core and (2) the attacker can reload/flush the data, checking access/flushing times, before the flushing phase. The attack results will be as follows: (1) if the data is in a cache, the attacker will think that it is the key data, however, the data could be data that the sensitive code loaded (the key data) or data that has been reloaded by SmokeBomb (not the key data); (2) if the data is not in a cache, the attacker will think that it is not the key data, however, the data could be data that the sensitive code did not load (not the key data) or data that has been flushed by SmokeBomb (the key data). As the possible attack results show, such attacks would have false-positive errors caused by SmokeBomb, and we believe that the attacks would be extremely difficult to trigger. Also, ARM CPUs do

not support a flush instruction except for ARMv8-A CPUs, and the flush instruction is typically not available to user-land applications. We note that, except for the previously discusses case, SmokeBomb can prevent all other cases of asynchronous attacks.

4.9.3 Case Studies

Case 1: OpenSSL—AES algorithm

The AES implementation of the OpenSSL library is a well-known target for cache side-channel attacks targeting its T-tables [79, 133]. AES T-tables are pre-computed lookup tables used to get a round key for each round of the AES algorithm. There are four 1 KB T-tables, for a total of 4 KB of sensitive data. If the secret key length is 128-bits, AES encryption and decryption processes have 10 rounds and the key is expanded into 10 round keys as well. This key expansion uses lookups against the T-tables, and determining these lookups via a cache side-channel allows an attacker to recover key data. We used two well-defined attack methods for this experiment: the last-round attack [79] and the one-round attack [133]. With the last-round attack method, it is possible to recover the full secret key. For the one-round attack, we can recover 4 bits of every key byte, since our experimental devices have a 64-byte cache line.

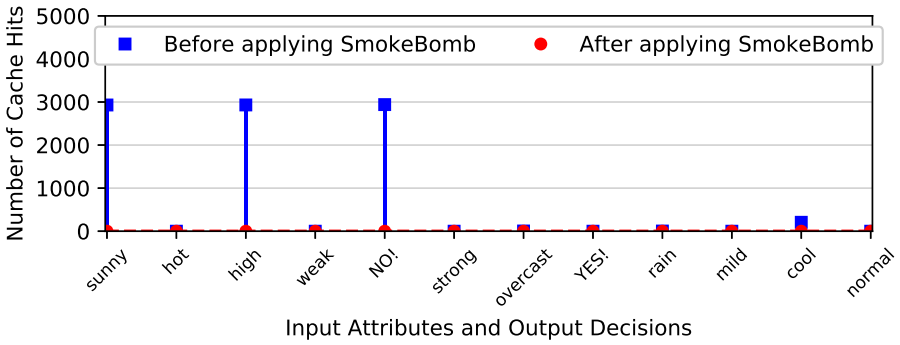
We first performed the last-round attack [79] and the one-round attack [133] without SmokeBomb. The attacks were conducted using the 128-bit AES algorithm (version 1.0.2 of the OpenSSL library). To demonstrate the effectiveness of SmokeBomb, we annotated the 4 KB T-tables of the OpenSSL library as the sensitive data (requiring four lines of code to annotate each of the T-tables), then SmokeBomb was applied to the library automatically.

In the last-round attack scenario, the attacker checks the cache state before and after the victim process executes the AES encryption function. Figure 4.5a and 4.5b show the attack results of Evict+Reload and Flush+Reload. The attacker can distinctly identify the addresses accessed by the victim without SmokeBomb. Without SmokeBomb's defense, we successfully recovered the secret key after 150 iterations of the attack. However, after SmokeBomb was applied, the attacker cannot observe any timing differences for all entries of the T-tables on both test devices, as shown in Figures 4.5a and 4.5b.

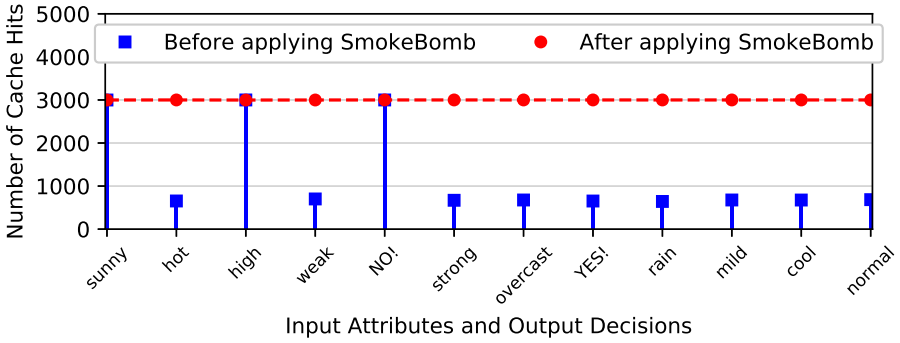
Similarly, Figure 4.5c shows the Prime+Probe attack results, in which we also can identify memory addresses accessed by the victim. The protection results in Figure 4.5c seem (to the attacker) to indicate that every entry of the T-tables was accessed by the victim. This is because *all cache sets where the sensitive data can be loaded have been occupied in the preloading step.* Thus, the attacker cannot understand *what data was accessed* and *cannot differentiate key data from sensitive data.* We ran the attacker and the victim processes on different cores concurrently.

To simulate the one-round attack, we sent a signal from the victim to the attacker process so that the attacker can perform the Flush+Reload attack after the first round of the AES encryption function—the attacker flushed the sensitive data before the `init` API executes and reloaded the sensitive data after the first round. Also, to avoid unnecessary impediments for conducting the attack, we paused the victim process before the second round—in the middle of the sensitive code execution. This makes *defense more difficult* by giving the attacker an advantage. With the one-round attack, we cracked half of the secret key over 500 iterations of AES encryption, on average. Figure 4.5d shows that the attack can reveal the T-table entries used in the first round of the AES encryption function.

Conversely, attacks against the SmokeBomb-protected implementation resulted



(a) Attacking Decision Tree on Cortex-A72.



(b) Attacking Decision Tree on Cortex-A53.

Figure 4.6: The Attack and Protection Results on the Decision Algorithm: EVICT+RELOAD was used on the Cortex-A72 and FLUSH+RELOAD was used on the Cortex-A53.

in cache hits for *all* entries—successfully protecting the sensitive data against attacker measurement. This prevention result shown in Figure 4.5d implies that the key data can be revealed by the attack using the directory protocol *unless* SMOKEBOMB preserves the sensitive data. After the first round is finished, the victim process still holds the sensitive data in the L1 cache but not in the L2 cache for executing the next rounds. However, access times to the sensitive data must be faster than the L2 cache miss by means of the directory protocol. Thus, the attacker has no choice but to think there was an L2 cache hit—the victim process has used that data.

Case 2: Decision Tree

A decision tree algorithm is used to make a decision according to some input data (called the attributes). Parameters are attributes and the output is the algorithm's decision. Each node of the decision tree is a point where an attribute is tested and a branch is taken according to the result of the test. A leaf node represents a final decision made by the attributes. Because different memory addresses are accessed depending on the attribute, this can result in information leakage via cache side-channel attacks [109].

For this experiment, we created a decision tree using the ID3 algorithm [63]. We also implemented a shared library which provides a service using the decision tree. The attack scenario is as follows: the victim calls the function within a shared library to get a result from the decision tree. To call the function, the victim needs to select specific information as attributes. A set of attributes is used as a parameter of the function. The attacker tries to identify the attributes selected by the victim and the decision made by the tree.

SmokeBomb was applied in the function that traverses the decision tree by annotating the tree as sensitive data (one line of code change). Because each of the different nodes tests the unique attributes and makes the final decision, there is a one-to-one correspondence between memory addresses of the nodes and attributes (or the final decision). Without protection, the attacker can clearly figure out the input records and the final decision as shown in Figure 4.6. SmokeBomb forces a consistent cache state for the sensitive data, and thus, the attacker cannot classify data as key data using access time. Figure 4.6b particularly shows the results of an attack in the middle of sensitive code execution. We simulated the attack to reload the sensitive data before the `exit` API executes. The attack results are all cache

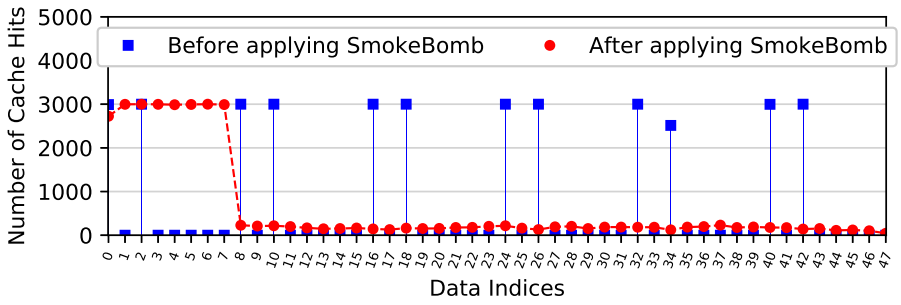


Figure 4.7: The Attack and Protection Result on the Large Sensitive Data: FLUSH+RELOAD was used on the Cortex-A53 processor. After applying SMOKEBOMB, we cannot find the access pattern.

hits because of the *directory protocol*, which indicates that the sensitive data is fully preserved.

Case 3: Large sensitive data

We show the effectiveness of SMOKEBOMB's defense when it protects application with sensitive data larger than L1 cache. In these experiments, the victim accesses 48 KB of sensitive data using a regular pattern and the attacker uses the FLUSH+RELOAD attack. The sensitive data consists of 48 entries and each data entry is separated by 1 KB (one line of code change to annotate the sensitive data). SMOKEBOMB selects only the first 8 KB as *selected sensitive data* if the sensitive data is larger than the L1 data cache. We conducted the attack in such a way that the attacker can check the data reloading time after the victim finishes accessing the sensitive data (before flushing it). Figure 4.7 shows the results: the attacker cannot infer the actual access pattern, only seeing cache hits on the first 8 entries and cache misses on the rest. This consistency protects against cache side-channel attacks *even when the sensitive data is larger than the L1 cache.*

4.9.4 Performance

We evaluate SmokeBomb's performance in a number of ways, from micro measurements to macro measurements.

Performance of software instructions

We evaluated the overhead of each `xSB` instructions emulated by SmokeBomb to keep the sensitive data in L1 cache (as discussed in Section 4.7). For the convenience of the experiment, we implemented a function that has only one instruction (either loading or storing data from/to a memory address) and measured its execution time in nanoseconds. We cannot measure CPU cycles directly because the cycle counter does not increase while execution is in the exception handler. Table 4.2 shows the execution times of `xSB` instructions on average across 5,000 executions with execution times of the orginal instructions.

Execution times of `xSB` instructions are substantial, when compared to the original `LDR` or `STR` instructions' execution times. Naturally, the more `xSB` instructions that execute, the larger the resulting performance overhead. SmokeBomb handles the performance overhead of executing `xSB` instructions by patching *only the cache changing instructions in the sensitive code that accesses* non-sensitive *data*. This optimization process helps to avoid unnecessary performance degradation. Using the decision tree (case study 3), the performance overhead when SmokeBomb patched *all* cache changing instructions increases by about 104% compared with the optimized patching.

Table 4.2: Comparison of the Execution Times Between Xsb Instructions and Original Instructions.

CPU	Instruction Group	L1 Hit		Cache Miss	
		Original	xSB	Original	xSB
Cortex-A72	LDR	612 *ns*	1,634 *ns*	897 *ns*	1,946 *ns*
	STR	622 *ns*	1,678 *ns*	729 *ns*	1,802 *ns*
Cortex-A53	LDR	321 *ns*	1,209 *ns*	480 *ns*	1,916 *ns*
	STR	365 *ns*	1,251 *ns*	540 *ns*	1,420 *ns*

Performance of SmokeBomb APIs

We evaluated the execution times of SmokeBomb API enter and exit APIs (which involves prefetching and flushing the sensitive data). As the performance overhead caused by SmokeBomb APIs is determined by the size of sensitive data, we measured execution times using different sizes of sensitive data up to a size the same as the L1 cache size (sensitive data sizes larger than the L1 cache size will only load the selected data to the L1 cache, therefore the upper bound is L1 cache size). Figure 4.8 shows the execution times of SmokeBomb APIs. The execution time of each API increases with the size of the sensitive data. When the size of sensitive data is 32 KB, the execution time of the two APIs is about 450 microseconds in total on the Cortex-A53.

Single-application overhead

To understand the impact of SmokeBomb on the performance of sensitive code, we evaluated a SmokeBomb protected HTTP Secure (HTTPS) protocol implementation. For this experiment, SmokeBomb was applied on the AES algorithm, and we used the top 500 web pages selected by the Moz [15]. Then, we compared av-

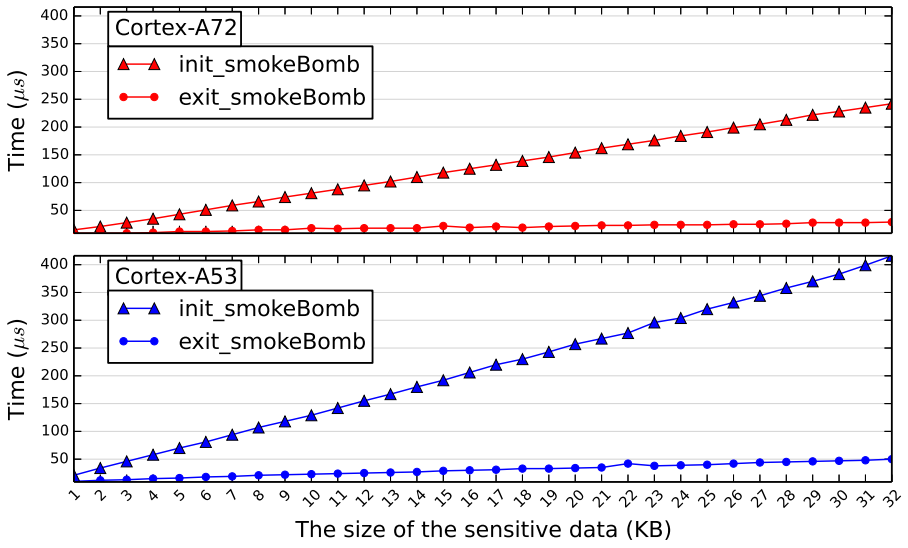


Figure 4.8: The Execution Times of SmokeBomb APIs in Microseconds.

erage execution times required to download the 500 web pages between the normal HTTPS protocol and SmokeBomb-protected one across 5,000 experiments. Table 4.3 shows SmokeBomb-protected HTTPS protocol has very low performance overhead of around 4.02% for Cortex-A53 and 5.91% for Cortext-A72, making it unnoticeable to users during web browsing and quite acceptable for serving web content.

We also applied SmokeBomb on the AES algorithm of 7zip application and measured execution times required to compress various files secured with AES encryption. As shown in Figure 4.9, on the Cortex-A53, the latency increased by just 1.58 percent, when 10 KB file is used. However, as the size of the input file increases, the latency also increases. The AES algorithm is a block cipher, and thus, its encryption function to which SmokeBomb was applied operates on a single block. Consequently, the performance overhead brought by SmokeBomb APIs and `xSB` instructions has to be overlapped as an input file has more blocks.

Table 4.3: The Performance Overheads of SmokeBomb-protected HTTPS to Load a Web Page.

Cortex-A72		Cortex-A53	
Baseline	SmokeBomb (overhead)	BaseLine	SmokeBomb (overhead)
7,223 *ms*	7,650 *ms* (5.91%)	7,177 *ms*	7,466 *ms* (4.62%)

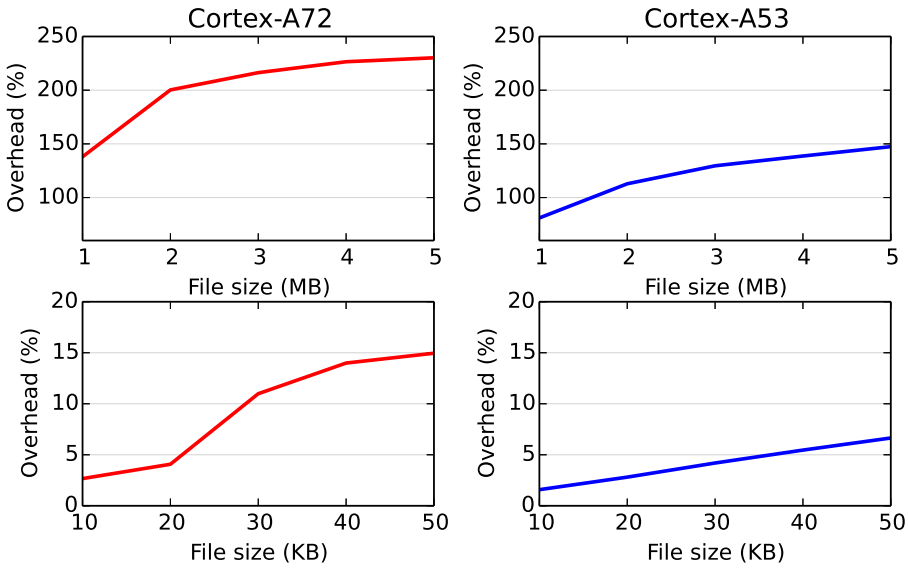


Figure 4.9: The Performance Overheads of SmokeBomb-protected 7zip Application.

Performance Impact on Systems

Lastly, we evaluate the impact of SmokeBomb on systems throughput by comparing: (1) when SmokeBomb is not running; with (2) when SmokeBomb is *running continuously* by a process executing the AES encryption function with SmokeBomb applied. The only difference between the two situations is the status of SmokeBomb functions in the testing systems. Thus, it can show how the system performance changes when SmokeBomb is activated. For the evaluation, we used the common UnixBench benchmarking utility (version 5.1.3) [14].

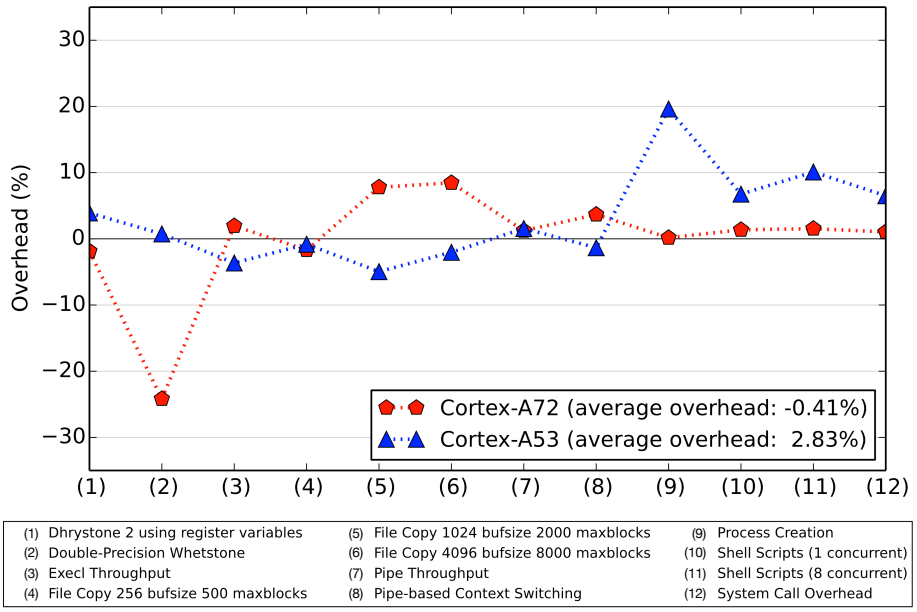


Figure 4.10: The Performance Overheads on Each Benchmark Applications When SmokeBomb Is Activated.

Figure 4.10 shows the overheads of each benchmark application and the average on the test devices. On the Cortex-A72 of a Samsung Tizen device which uses the inclusive cache model, and where many service processes are executing, the average overhead with SmokeBomb is almost zero. We suspect that the negative overhead of the Whetstone benchmark, which tests how many floating-point operations can execute within a limited time (without using a blocking syscall), is because the benchmark threads could occupy cores a longer time than when SmokeBomb is not running. The average overhead on the Cortex-A53 of Raspberry PI3, which uses the non-inclusive cache model, is about 2.8%. The results illustrate that, when SmokeBomb is activated, the average performance overhead that SmokeBomb imposes on the overall system is negligible.

4.10 Limitations

This section discusses some of SmokeBomb's limitations, which can help inform future research directions.

Instruction cache attacks

The first limitation of SmokeBomb is that the *instruction cache* is not protected, potentially allowing an attacker to understand which instructions are executed by the victim process (though not what sensitive data was accessed). There is a fundamental problem in applying SmokeBomb to fully protect the instruction cache: ARM has a preloading instruction (PLI) instruction, but in practice we cannot use this instruction to fetch instructions into the L1 instruction cache. The effect of the PLI instruction is not explicitly defined in the ARM architecture reference manuals [27, 31]. The pre-loading instruction (PLI) instruction is treated as a NOP instruction in several Cortex-A processors [40, 41, 33, 34, 35], or it fetches instructions to the L2 cache instead of the L1 instruction cache [42]. Thus the only general way to load the L1 instruction cache is to execute the instructions.

However, SmokeBomb can be extended to have functionality for preloading instructions, and flush them, from L2 cache. Given that a combination of preloading and flushing is enough to prevent synchronous instruction cache attacks, a future direction is to defeat asynchronous attacks that inspect an instruction cache in parallel with a running victim process. Note that preloading and flushing make this kind of attack very difficult to perform. In addition, most ARM Cortex-A processors employ a virtually indexed, physically tagged (VIPT) implementation for the L1 instruction cache [32, 26, 21, 24, 28, 40, 41, 33, 36, 43]. Therefore, it is a reasonable assumption that instructions are safe from attacks using the directory protocol because modern

operating systems implement ASLR. Aside from this limitation in terms of attacks to determine the execution of sensitive *code*, SmokeBomb can stop known attacks with respect to the sensitive *data*.

Protection for exclusive caches

SmokeBomb cannot provide complete defense for exclusive caches. This is because if an attacker loads the sensitive data into their L1 cache, and evicts the data from the L1 to the L2 cache, the data could also be evicted from the victim's L1 cache. By the definition of the exclusive cache, there is only one copy of the data in the whole cache. Therefore, if an exclusive L2 cache is shared by multiple cores, cache lines of the L1 cache could be affected by the other cores' data usages. As a result, the attacker might deduce information related to the key data. Such information leaks are not theoretically impossible but are difficult to practically achieve, because an attacker must identify whether the data is reloaded by a victim after evicting it within a very short execution time between two SmokeBomb APIs. Moreover, the reloaded data may not be the key data. Unfortunately, we could not conduct any experiment on the exclusive cache model, because there are no available devices on the market. Only the Cortex-A55 employs the exclusive L2 cache among ARM CPUs, but, that is the *private per-core* unified L2 cache which is not shared between cores [42].

Implementation details

The instruction handler might *evict some sensitive data from the L1 to the L2 cache* as the size of sensitive data reaches the size of L1 cache. We evaluated L1 cache refill events caused by the instruction handler during its execution and determined that three cache lines were used by the instruction handler due to the accesses to the stack

and global variables. Such evictions could be handled by a dynamically allocated temporary stack only for the execution of sensitive code. This temporary stack can, then, be cached into the L1 data cache with the sensitive data to prevent any eviction of the sensitive data at a small cost of the private area. We note that even if the eviction of sensitive data occurs it would not be critical to SmokeBomb's effectiveness, as the attacker's detecting of eviction does not necessarily lead to the leakage of key data. In addition, the current version of the SmokeBomb implementation protects static data. However, SmokeBomb could provide the same defense for dynamically allocated data with additional improvements to the compiler extension, which would require annotations on pointers that are used to point to sensitive data. In addition, albeit SmokeBomb only requires an annotation of the sensitive data, it has to recompile source code, and thus, cannot be applied to compiled binaries. Lastly, the implementation of SmokeBomb is dependent on hardware specifications such as the size and inclusiveness of the cache. Hence, minor changes are required to implement SmokeBomb for each different CPU.

Architecture dependence

While the concept of a *private space* in L1 cache is not ARM-specific, SmokeBomb is implemented for, and heavily uses specific functionality of, the ARM architecture. Unfortunately, in our investigation of the current state of the Intel x86_64, architecture, it does not appear to be possible to ensure that data is present *only* in L1 cache, preventing us from implementing SmokeBomb for this architecture. Likewise, the RISC-V architecture currently has no instruction-level control of the cache [141]. However, as both cache control and hardware-based security measures are an actively-evolving field, this could change in the future.

4.11 Related Work

Because the shared feature of hardware resources is one of the fundamental reasons behind cache side-channel attacks, many proposed countermeasures attempt to isolate shared resources to mitigate such attacks. These countermeasures can be categorized as a hardware approach or a software approach. Previously, most of the software approaches are deployed on cloud systems with Intel architecture. SmokeBomb is the only cache side-channel defense without architecture-specific hardware dependencies, covering the L1 and the L2 cache together. Thus, SmokeBomb is the first defense applicable to the ARM architecture. It achieves this without invasive OS changes. Furthermore, it can be applied to applications automatically by annotating the sensitive data.

SmokeBomb's closest related works are as follows. Kim et al. [84] proposed isolation of the last level cache using a dedicated memory page on each core. Even though it can prevent information leakage via the last level cache efficiently, this approach cannot prevent timing attacks using upper-level cache [78]. Zhang and Reiter [165] proposed periodic cache cleansing mechanism, which prevents information leakages by flushing data used by the previous process in the cache. It cannot address cache attacks targeting the last level cache. Zhou et al. [167] introduced the copy-on-access technique which copies the page when another process accesses memory simultaneously to disable memory sharing. Also, it limits the cacheability of memory pages per process, and thus, each process only can have a limited number of cache lines. Liu et al. [93] presented CATalyst which uses the Intel Cache Allocation Technology to partition the last level cache. It disallows sharing the cache as in STEALTHMEM [84] to defeat the last level side channel attack.

Most recently, Gruss et al. [69] proposed a technique that uses hardware transac-

tional memory (HTM) to prevent cache misses during execution. Though it provides strong cache side-channel protection, the protection range is limited by the size of the CPU's caches. Gruss et al.'s approach requires hardware support, and the ARM architecture does not support the HTM. Another concern is the possibility of these protected transactions failing, which happens frequently under heavy system load (and could be induced by attackers and result from attacks) [69].

4.12 Conclusion

We presented SmokeBomb: a novel, systematic software approach to defeat cache side-channel attacks on the ARM architecture. Our mitigation approach protects access patterns on the sensitive data from attackers easily by providing the protection mechanism to applications as a compiler extension. Our experimental results show that SmokeBomb protects sensitive information leakages against cache attack methods known to us effectively—and with minimal overhead—on overall system throughput.

Chapter 5

EXPLOITING USES OF UNINITIALIZED STACK VARIABLES IN LINUX KERNELS TO LEAK KERNEL POINTERS

5.1 Introduction

For performance concerns, unsafe programming languages, such as C and C++, are still prevalently used in the implementation of operating system (OS) kernels and embedded systems. While these unsafe languages may allocate memory on stack or in the heap for variables, these variables may not be initialized before being used. When a variable is used without proper initialization (which can be caused by either a programming mistake or padding bytes in a struct inserted by compilers [138]), the memory values that were present at the same location of the variable before it was allocated—called *stale values*—will be read and used. When these stale values are copied from the kernel space to the user space, user-space programs will be able to access them, which causes an information-leak vulnerability if the information contained in the stale values is important.

The use of stale values in Linux kernels can lead to severe security problems, which have been studied in the past [54, 116, 97]. Moreover, these stale values can pose severe security threats without being directly used in the kernel. For example, modern kernel security defenses, such as Kernel Address Space Layout Randomization (KASLR), depend on keeping kernel addresses secret from user-space programs. When attackers get lucky and recover kernel pointer values through leaked information (stale values) from the kernel space, they can defeat KASLR [147, 96]. Likewise, attackers may leak cryptographic keys that are stored in the kernel space.

	Total	Stack-based	Heap-based	# of exploits
# of CVEs	87	76 (87%)	11 (13%)	0

Table 5.1: The number of information leak CVEs that are related to uses of uninitialized data between 2010 and 2019. The majority of these CVEs are stack-based information leaks. There are no publicly available exploits for these CVEs. Only one out of these 87 CVEs warns about possible leaks of kernel pointers and potential KASLR bypasses.

Unfortunately, in Linux kernel, information leaks that are caused by uninitialized data are common. A study shows that information leak vulnerabilities that are caused by the use of uninitialized data are the most prevalent type of vulnerabilities among the four major types of vulnerabilities in Linux kernel [50]. Within the past two years, KernelMemorySanitizer (KMSAN) discovered over 100 uninitialized data use bugs in Linux kernel through fuzzing [10]. Worse, due to the difficulty (or the impossibility) of exploiting the majority of information leak vulnerabilities or using them in high-risk exploits (such as remote code execution or local privilege escalation), these vulnerabilities are commonly believed to be of low risks. As a result, many uninitialized data uses do not get sufficient attention from developers or security researchers, are not assigned any CVE entries[1], and in some cases their corresponding patches are not merged into Linux kernel for a long time [97].

Table 5.1 shows the statistics of 87 Linux kernel CVEs that are related to uninitialized data uses and are reported between 2010 and 2019 [11]. The majority of

[1]Here is an example of a security patch that fixes a stack-based information leak vulnerability: `https://github.com/torvalds/linux/commit/7c8a61d9ee`. No CVE was ever assigned for the vulnerability.

these CVEs are stack-based information leaks. Evaluating the severity of these CVEs is extremely difficult since no public exploit is available for any of them. Even if a public exploit is available, using these vulnerabilities to leak key information usually requires manual and complicated manipulation of the kernel layout, which is costly and time-consuming. Therefore, all but one CVE (CVE-2017-1000410) mentions anything about the potential of leaking kernel pointers and bypassing KASLR, which leaves an impression to the general public that these vulnerabilities are of low security impact.

The situation about information leaks in Linux kernel is extremely concerning. In this work, we demonstrate the *actual* exploitability and severity of information leak bugs in Linux kernels by proposing a generic and automated approach that converts stack-based information leaks in Linux kernels into vulnerabilities that leak kernel pointer values. Specifically, we focus on leaking pointer values that point to kernel functions or the kernel stack. These leaked kernel pointer values can be used to bypass kernel defenses such as KASLR, which is an essential step in modern Linux kernel exploits [81].

Our proposed approach takes as input an exploit that triggers a stack-based information leak bug, analyzes the exploit to identify locations where stale values are coming from, and reasons about an attack vector that places kernel pointer values at these locations. It is worth mentioning that our approach supports leaking kernel pointers when the size of the leaked stale value is less than a full 64-bit pointer (8 bytes). We evaluate our approach on five real-world Linux kernel vulnerabilities (including four CVEs and one bug that was reported by KMSAN) and demonstrate its generality and effectiveness. The existing Common Vulnerability Scoring System (CVSS) scores of three of the above CVEs are 2.1 (on a scale of 0 to 10, higher is more severe), which imply that "specialized access conditions or extenuating circumstances

Criteria	Our approach	Lu *et al.* [97]	Xu *et al.* [147]	Halvar Flake [61]
Types of unused memory targeted for generating exploits	Stack	Stack	Stack, Heap	Stack
Generating exploits for leaking sensitive data	✓	✗	✓	✗
Finding locations of uninitialized data	✓	✗	✗	✗
Reasoning about storing sensitive data at a given location	✓	✓	✗	✗

Table 5.2: Comparison of Our Proposed Approach for Uninitialized Memory Uses with the Other Approaches: Although there are several prior research works that focused on exploiting uninitialized data uses (such as uninitialized pointer dereferences), there has been no research effort on exploitations of stack-based information-leak bugs for leaking kernel pointer values.

do not exist, even though there is considerable informational disclosure" [4, 3, 2]. Our findings can be used to assist CVSS in correcting the scoring and assessment of information leak vulnerabilities in Linux kernels, and raise awareness in the security community of these vulnerabilities.

5.2 Background

In this section, we introduce our goal comparing with prior research work and how leaked kernel pointer values can be used in more severe types of kernel exploits.

5.2.1 Uninitialized Data in Linux Kernel Exploitation

As shown in Table 5.2, prior research work mostly focuses on controlled uses of uninitialized data in Linux kernels [61, 147, 97]. Our work has a totally different goal: we focus on exploiting existing stack-based information leak vulnerabilities and converting them into high-impact vulnerabilities that leak sensitive data from the kernel. To the best of our knowledge, there is no prior research on the exploitation of stack-based information leak bugs in Linux kernel for leaking sensitive information.

Bypassing KASLR

Commonly used in OS kernels, KASLR is a defense mechanism that randomizes the base address of the kernel (where the kernel code is loaded) at boot time. This technique was introduced to raise the bar of kernel memory corruption attacks (*e.g.*, buffer overflows and use-after-free attacks) and is one of the most effective defenses in modern OS kernels. Systems with KASLR enabled can successfully mitigate memory corruption attacks as long as the attacker cannot learn randomized kernel addresses through information disclosure or side channel leaks [81]. A kernel pointer leak will naturally lead to the bypass of KASLR, which we will detail next.

The Linux kernel on x86-64 architecture implements 6 bits of entropy for the kernel code. The address range of kernel text section is 1 GB (`0xffffffff80000000` – `0xffffffffc0000000`) and the base address of kernel text is aligned by 16 MB. Hence, there are 64 virtual memory addresses (1 GB ÷ 16 MB) where the kernel `.text` section can be loaded. Consequently, on a condition that we can leak a kernel pointer value pointing to a kernel function, we will be able to calculate the KASLR slide-byte by simply subtracting the 5th byte of the leaked pointer value from the 5th byte of kernel text section's start address. As an example, if the leaked kernel pointer value is `0xffffffffa9a72cc0`, the KASLR slide-byte is `0xa9`−`0x80`=`0x29`. Attackers can compute randomized addresses of all kernel functions by using the slide-byte.

Attacking the kernel stack

Another type of kernel pointer values that we attempt to leak are pointer values that point to the kernel stack. These kernel pointers can be used to identify where the kernel stack is. The location of the kernel stack must not be discovered by at-

tackers because it is critical information that the attackers can use in their exploits to defeat KASLR and achieve arbitrary kernel code execution. For example, the kernel stack contains return addresses of kernel functions and values of the stack canary on which the entire stack overflow protection mechanism relies. Additionally, at the bottom of the kernel stack, the `thread_info` structure is stored (when `CONFIG_THREAD_INFO_IN_TASK` is disabled). This data structure includes architecture-specific thread-related information and a pointer to the `task_struct` that holds process-related information.

5.3 Attack Model

We assume that the attacker targets an x86-64 Linux system and tries to leak kernel pointer values that point to either kernel functions or the kernel stack. As previously discussed in Section 5.2.2, this step is very important for defeating modern kernel defenses, such as KASLR, before mounting future attacks.

To exploit information-leak bugs for leaking kernel pointer values, an in-depth analysis on the target kernel and the information-leaking bug is essential. Through this analysis, the attacker obtains critical information for exploiting the vulnerability, such as what types of kernel pointer values can be leaked, and where to place the kernel pointer values. We assume that the attacker has access to a local machine with the same Linux kernel and configuration as the target system, which the attacker can use to conduct the analysis and perform the attack before launching it on the target system. The attack should have full access to the local machine. We also assume that the attacker possesses the required exploit that triggers the information leak, which, at this moment, is likely to not leak any sensitive information on the kernel stack. With the analysis results, the attacker will generate exploits that can execute on the target system without the root privilege and reliably leak kernel pointer values.

5.4 Challenges in Exploitation

We demonstrated how kernel information leaks can occur via uninitialized stack uses with Listing 2.1 and Listing 2.2. However, simply triggering the vulnerabilities will most likely not copy any sensitive data from the kernel stack to the user space. Therefore, we must be able to manipulate data on the kernel stack and ensure kernel pointer values (or part of a kernel pointer value) are put in uninitialized variables on the stack. To this end, we must analyze each vulnerability and generate a proper exploit for it, which involves tackling the challenges that Lu, *et al.* previously discussed [97]. It is worth mentioning that our goal is different from theirs, and thus, we define a series of challenges that we must overcome to successfully leak kernel pointer values as follows.

C1: Computing the offset to uninitialized data from the kernel stack base.

The first challenge to leaking kernel pointer values is identifying the distance to an uninitialized memory cell from the base address of the kernel stack, which we term *leak offset*. Computing the leak offset allows us to find the exact location where kernel pointer values should be stored. We identify the leak offset through applying a new technique, called *footprinting*, on the kernel stack in Section 5.5.1.

C2: Storing kernel pointer values at a leak offset.

The next challenge is finding a way to place kernel pointer values at the specific leak offset. To achieve this goal, we propose two methods: (1) syscall enumeration with the help of the Linux Test Project (LTP) to find syscalls that can be used to store kernel pointer values at the leak offset (see Section 5.5.2), and (2) kernel stack spraying using the extended Berkeley Packet Filter (BPF)

(see Section 5.5.3).

C3: Handling data leaks that are less than 8 bytes.

On a 64-bit Linux kernel, when a kernel data leak is larger than 8 bytes, we can obtain the value of the whole pointer. However, in many vulnerabilities, the size of memory leak is smaller than 8 bytes—we cannot obtain a complete pointer value. For handing such small leaks, we reason a possible range of the unleaked value through the guess and check method, by which we can identify the base address of the stack kernel. We discuss about the small leaks in Section 5.5.4.

Ideally, in addition to the above challenges, we should also prevent any future overwriting to kernel pointer values that we stored in the stack before the data is copied to the user space. This is to guarantee the successful exploitation of information-leak bugs. Unfortunately, there is no practical method to prevent such unexpected data overwriting without hijacking the control flow of the kernel on the target system. Thus, in this work, we consider such cases where stored stack values are later overwritten before returning to user space to be *unexploitable*.

5.5 Exploiting Uninitialized Stack Variables

Our goal is to design a generic approach to exploit stack-based information-leak vulnerabilities for leaking kernel pointer values. In the rest of this section, we describe how we tackle the challenges that are represented in Section 6.5.

5.5.1 Computing the Leak Offset

We propose a novel technique, called byte-level stack footprinting, to identify the distance to an uninitialized memory address from the base address of the kernel stack. The mechanism is illustrated in Figure 5.1.

First, we write offset information to each byte of the stack from the base address by hooking a syscall. In 64-bit kernels, the kernel stack is 16-byte or 8-byte aligned at a new frame of a function starts and every pointer in the stack is stored at 8-byte aligned addresses. We store 1-byte offset information which starts from `0x0` to `0xff` in each byte for every 8 bytes. Therefore, even though with 1-byte information leak, we can identify exact offsets at which kernel pointer values should be stored to leak them. This mechanism allows us to footprint 2,024 bytes of the kernel stack. Even though we cannot footprint the entire kernel stack, 2,024 bytes are enough to deal with most syscalls (roughly 90% of syscalls only use less than 1,260 bytes of the stack).

Then we trigger an information-leak vulnerability. Because the offset information has been filled into the stack, we can directly check the offset. Lastly, we compute a leak offset by using the offset information from the kernel. For example, in Figure 5.1, the offset information copied from the kernel is `04`, and thus, we need to find kernel pointer values that can be stored at an offset ($Base - 24$).

5.5.2 Extensive Syscall Testing with the LTP

Once the leak offset has been identified, we need to find a syscall and its arguments that can be used to store a kernel pointer value at each leak offset. For fast and reliable testing, we leverage the Linux Test Project (LTP) which provides various tools and concrete test cases for syscalls [88]. We supplement three additional steps onto each syscall test case in LTP: (1) spraying the stack with a magic value; (2) finding kernel pointer values stored in the stack; and (3) recording context information.

Figure 5.2 shows how our syscall testing framework finds proper syscalls and arguments. Before executing a syscall, we fill the kernel stack with a magic value to detect data changes that are made by the execution of the syscall. Then we inspect the kernel stack from the base address to find kernel pointer values. To this end, we

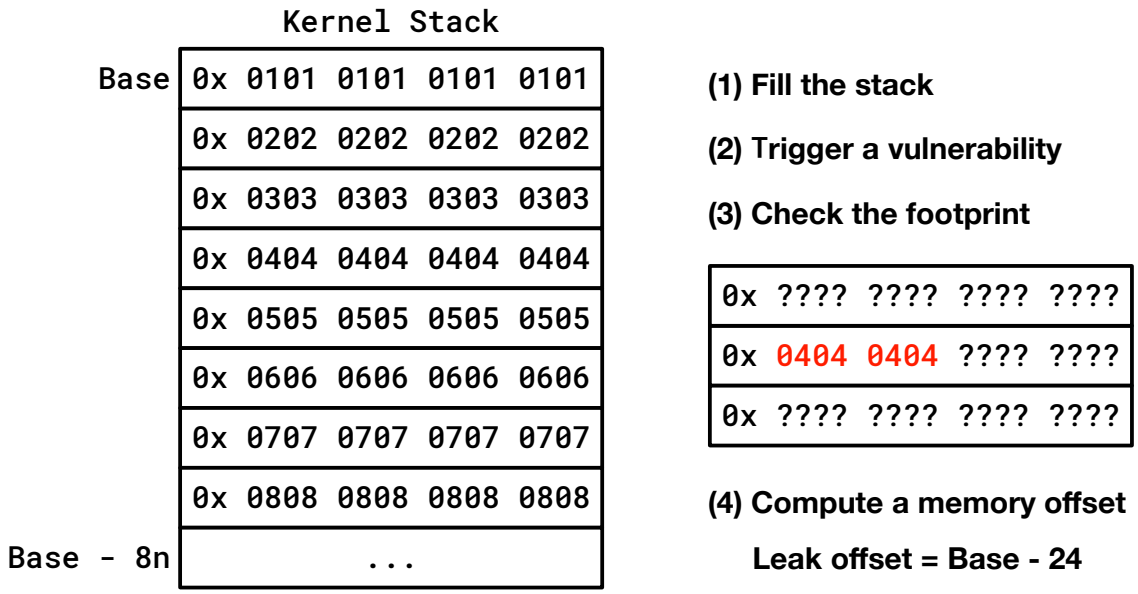


Figure 5.1: The Footprinting Mechanism to Compute a Leak Offset: With this mechanism, we can footprint 2,024 bytes of the stack (about 90% of syscalls use less than only 1,260 bytes of the stack).

check every 8-byte from the stack base whether each 8-byte value is in the address range of the kernel stack or the kernel code region (the `.text` section). If we find any kernel pointer value that points to the kernel stack or kernel code, we record the name of the syscall with its arguments and pointer type. From the recorded information, we select proper context data (a syscall and arguments) that can store a kernel pointer value at a specific leak offset.

5.5.3 Stack Spraying via BPF

Designed to support filtering packets as requested by user-space applications, the extended Berkeley Packet Filter (BPF) is a virtual machine that resides inside the kernel [100]. The BPF virtual machine takes as input BPF programs (that use a special instruction set) when a user-space application attaches the BPF program

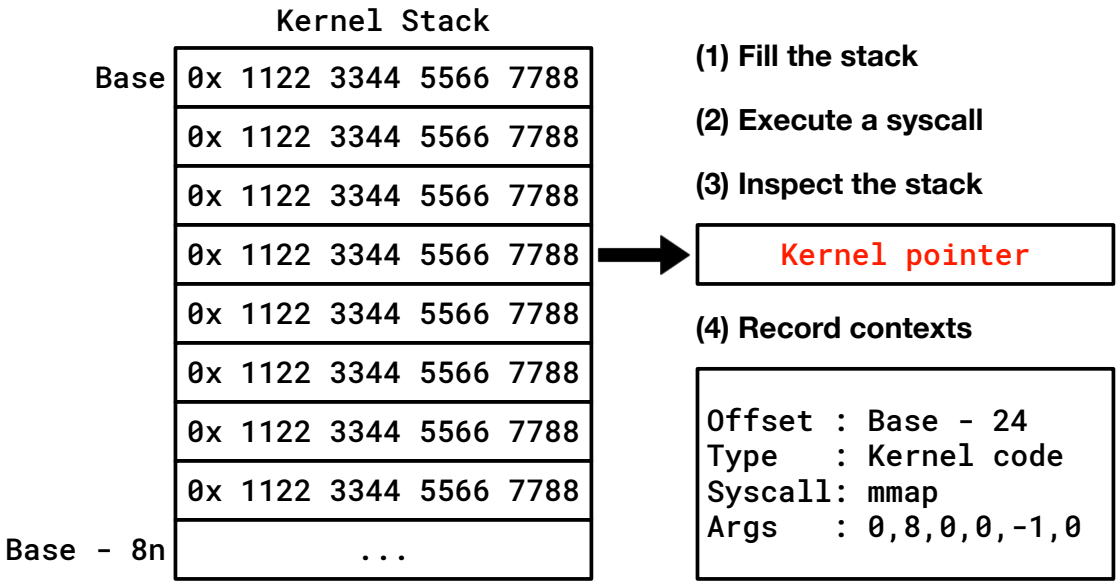


Figure 5.2: The Routine of Our Syscall Testing Framework: We first fill the stack with a magic value. Then we execute a syscall and find kernel pointer values stored in the kernel stack. Lastly, we record the context information.

onto any socket. Then, the BPF program executes when data passes through its attached socket and filters data as programmed.

BPF programs can use stack memory, which is allocated inside the kernel stack. Listing 5.1 shows the first part of the `bpf_prog_run` function, which shows that the stack of any BPF program is limited to 512 bytes. In the BPF virtual machine, there is a special register, `R10`, called the *frame pointer*. This register points to the top of the stack (the stack base) that a BPF program uses. Therefore, the frame pointer always points to a location on the kernel stack. we use this frame pointer and the location of the stack of a BPF program to spray the stack kernel.

With a carefully crafted BPF program, we can store the frame pointer value to the stack of a BPF program until the stack is full. In other words, we can store an address of the kernel stack up to 512 bytes inside the kernel stack. Additionally, the

```
1 static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
2 {
3   u64 stack[MAX_BPF_STACK / sizeof(u64)];
4   // 512-byte stack for a BPF program
5
6   u64 regs[MAX_BPF_REG], tmp;
7   ...
```

Listing 5.1: The main function for executing a BPF program. It allocates the stack for BPF programs and execute them.

location of the `stack` local variable of the `bpf_prog_run` function changes depending on functions previously executed. Therefore, different execution paths *from* various syscalls transmitting data using a socket *to* the `bpf_prog_run` function can change the stack spraying range (discussed in Section 5.6.3 with a case study).

Listing 5.2 shows a part of a BPF program that sprays the kernel stack with the frame pointer. On Line 20, we copy the frame pointer (`R10`) to the `R3`. From Line 21, we spray the stack of a BPF program (kernel stack) with the frame pointer. It is worth noting that BPF virtual machine strictly restricts behaviors of a BPF program for preventing security issues by using the static verifier [12]. As examples of the restrictions for every BPF program, all memory access is bounded, there cannot be unreachable instructions, the frame pointer (`R10`) is a read-only register and so forth. However, the static verifier allows our BPF program to execute stack spraying. we manually inspected the verifier and could not find a rule for preventing spraying the frame pointer.

If we can store the frame pointer at a leak offset through the stack spraying, we will be able to figure out where the kernel stack is. Moreover, we can learn the memory layout of the kernel stack when a syscall executes based on the location of the kernel.

```
#define BPF_MOV64_REG(DST, SRC)                 \
((struct bpf_insn) {                            \
  .code  = BPF_ALU64 | BPF_MOV | BPF_X, .dst_reg = DST, .src_reg = SRC, .off   =
    0, .imm   = 0 })
#define BPF_STX_MEM(SIZE, DST, SRC, OFF)        \
((struct bpf_insn) {                            \
  .code = BPF_STX | BPF_SIZE(SIZE) | BPF_MEM, .dst_reg = DST, .src_reg = SRC, .
    off   = OFF, .imm   = 0 })
void stack_spraying_by_bpf() {
  struct bpf_insn stack_spraying_insns[] = {
    BPF_MOV64_REG(BPF_REG_3, BPF_REG_10),
    ...
    BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3, -392),
    BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3, -400),
    BPF_STX_MEM(BPF_DW, BPF_REG_10, BPF_REG_3, -408),
    ...
  };
  ...
```

Listing 5.2: A code snippet to perform kernel stack spraying using BPF. we can spray the frame pointer of a BPF program for 512 bytes on the kernel stack.

5.5.4 Handling Small Data Leaks

If an information-leak vulnerability leaks 8 bytes or more than 8 bytes of data, and we can store a kernel stack address at a leak offset through spraying the stack with a BPF program, it is possible to fully recover a kernel stack address. Unfortunately, the sizes of leaks of many stack-based information-leak vulnerabilities (roughly 60% of them) are smaller than 8 bytes [11]. Because the kernel stack is aligned by the size of a page (*e.g.*, 4KB by default), we need the most significant 52 bits of a kernel stack address (a 7-byte leak) to get the base of the kernel stack. Therefore, leaks that are smaller than 7 bytes cannot be directly used to reveal the kernel stack base.

To handle this problem, we investigated the static verifier of the BPF virtual

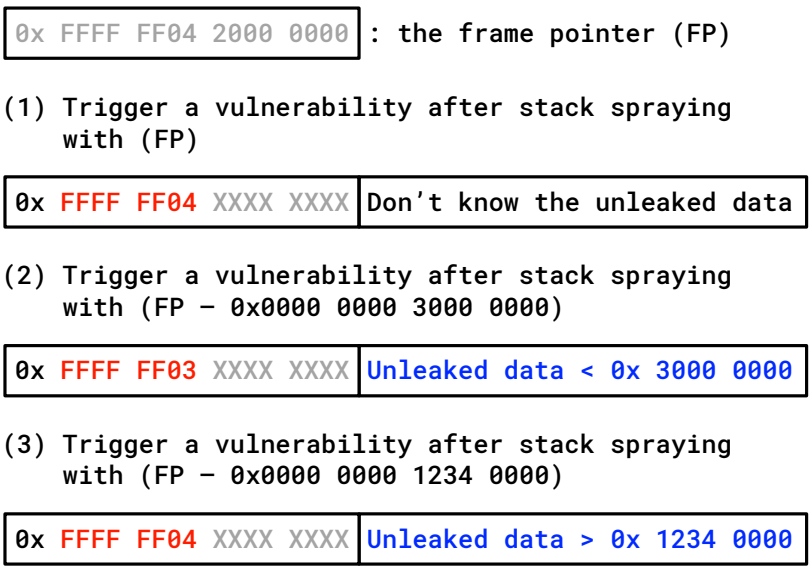


Figure 5.3: The Procedure for Identifying a Kernel Stack Address Using the Kernel Stack Ppraying via a BPF Program: By checking changes of the leaked data, we figure out possible ranges of the unleaked data until the stack base address is revealed.

machine to check if arithmetic operations on the frame pointer is possible. We found that some arithmetic operations (such as bitwise shift) are not possible, but `add` and `sub` can be used with arbitrary immediate values. If the BPF allowed bit shifting operations on the frame pointer value, we would simply shift the frame pointer value so that unleaked data can be placed at leak offset. We can only execute `add` and `sub` operations on the frame pointer. However, these operations can be executed even if the result is beyond the range of the kernel stack. We also found that, after executing these arithmetic operations, the modified frame pointer value can be stored at the kernel stack.

By using this unrestricted behavior of a BPF program, we deal with such small leaks using the guess and check method to identify unleaked data of a kernel stack address, and, eventually, to reveal the layout of the kernel stack. This strategy requires manipulating the frame pointer value and check how known (leaked) data

changes. Figure 5.3 illustrates how a 4-byte information leak vulnerability can be used to identify the base of the kernel stack by reasoning it. we first trigger a vulnerability after spraying the kernel stack with the frame pointer. Next, we execute an arithmetic operation (`add` or `sub`) on the frame pointer with an arbitrary immediate value. This modified frame pointer value is sprayed and we check the leaked data by triggering the vulnerability. As shown in Figure 5.3, when we sprayed (`FP−0x30000000`), the leaked data has changed from `0xffffff04` to `0xffffff03`, by which we can notice that the frame pointer value is smaller than `0xffffff0430000000`). We repeat this reasoning procedure until we can obtain the kernel stack base address: until the most important 52 bits of a kernel stack address is revealed.

We note that, a security patch was applied to the upstream Linux kernel at April 18th 2019 from the version 4.14.113[2] to restrict arithmetic operations on the frame pointer for unprivileged users so that the frame pointer value cannot go out of the stack region.

5.6 Evaluation

We evaluate our proposed approach against real-world information leak vulnerabilities in Linux kernels that involve uses of uninitialized stack variables. In this section, we first present the implementation of our tool in Section 5.6.1, then present the evaluation results of our syscall-enumeration-based pointer finding approach (in Section 5.6.2), and finally present case studies of all five vulnerabilities that we evaluated against (in Section 5.6.3).

[2] `https://lore.kernel.org/patchwork/patch/1063913/`

5.6.1 Implementation

We implemented an analysis tool, which consists of a shared library (KptrLib) and a loadable kernel module (KptrMod), to automatically find leak offsets (as described in Section 5.5.1). Then, we modified the Linux Test Project (LTP) to perform the three additional steps using KptrMod, as discussed in Section 5.5.2. We also implemented a tool for automatically spraying the kernel stack and handling small leaks with a BPF program (as described in Section 5.5.3 and Section 5.5.4).

Our tools can be easily used to analyze any given exploit that triggers a information-leak vulnerability to evaluate its exploitability regarding identifying the location of the kernel stack, leaking kernel pointers, and finally bypassing KASLR.

5.6.2 Finding Pointers with the LTP framework

First, we evaluate the effectiveness of our syscall enumeration framework. To this end, we ran the modified LTP on Ubuntu 18.04 (with Linux kernel v4.15.0). For each kernel, we need to run the LTP framework once to record context information. Then, we can simply pick a context (a syscall and its argument) from the recorded data for storing a kernel pointer value at the identified leak offset.

Figure 5.4 illustrates that how many contexts (combinations of a syscall and its arguments) can store sensitive pointer values (pointing to the kernel code or stack) for each stack memory offset less than 2,298. The experimental results show that our modified LTP framework can find syscalls to store kernel pointer values at almost every stack offset when offsets are larger than `the stack base + 440`.

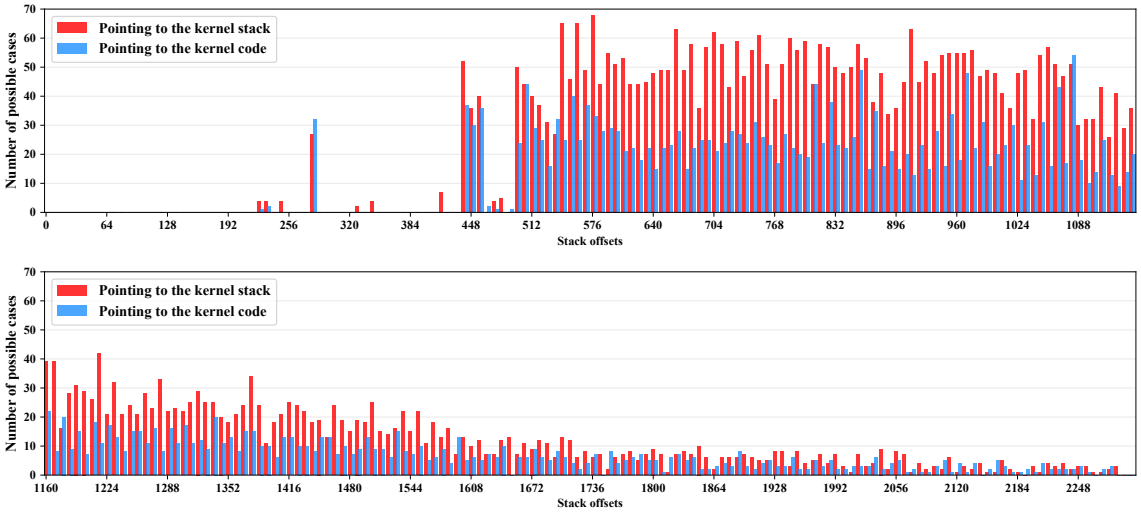


Figure 5.4: The Experimental Result of the Modified LTP Framework: We can find syscalls to store kernel pointer values at almost every stack offset, when offsets are larger than 440 bytes and smaller than 2,298 bytes, through the LTP framework.

5.6.3 Case studies

To evaluate our approach, we select four CVEs and one fixed bug (which a CVE entry has not been assigned) in the Linux kernel and generate an exploit for each vulnerability according to our analysis results.

Why not analyze more CVEs?

While we agree that analyzing more CVEs will help better demonstrate the generality and applicability of our proposed approach, during the evaluation of our approach, we realized that it is extremely time-consuming to develop exploits for these CVEs to reliably trigger the intended information-leaking bugs. Thus, we deem it infeasible to evaluate our approach on more CVEs for which no public exploits are available. We believe that these randomly selected CVEs are covering all scenarios that an attacker may face and are sufficient for demonstrating the generality of our proposed approach.

CVE-2018-11508

As we have shown in Section 2.3, this vulnerability is caused by an uninitialized stack variable (`tai` field of the `txc` struct). The CVSS score of this vulnerability is 2.1 [4]. We speculate that the impact of this vulnerability is deemed low because the size of information leak is only 4 bytes, which are not enough to host an entire pointer on 64-bit Linux systems.

After checking a leak offset through KptrLib and KptrMod, we found that this vulnerability leaks 5th byte to 8th byte of a pointer value. Also, we confirmed that a pointer value that points to the kernel text can be stored at the leak offset from the dataset recorded by the modified LTP in Section 5.6.2. Consequently, we can successfully get the KASLR slide from this vulnerability.

CVE-2016-4569 and fix 372f525

The CVSS score of CVE-2016-4569 is also 2.1. Our proposed approach successfully exploits this bug and identifies the KASLR slide.

The patch (fix 372f525) did not become an official CVE entry. In the commit message of the patch, a kernel developer mentioned that "*There should be no danger of breaking userspace as the stack leak guaranteed that previously meaningless random data was being returned.*[3]" Unfortunately, our proposed approach works against this bug and successfully identifies the KASLR slide. This demonstrates the necessity of our approach for proving the severity of information leak bugs in Linux kernels.

CVE-2016-4486

With this CVE (CVSS score 2.1), we show that how a 4-byte leak vulnerability can be exploited to identify the kernel stack base.

[3] `https://github.com/torvalds/linux/commit/7c8a61d9ee`

```
1 /*file: net/rds/recv.c */
2 void rds_inc_info_copy(struct rds_incoming *inc, struct rds_info_iterator *iter,
      __be32 saddr, __be32 daddr, int flip)
3 {
4   struct rds_info_message minfo;
5   minfo.seq = be64_to_cpu(inc->i_hdr.h_sequence);
6   minfo.len = be32_to_cpu(inc->i_hdr.h_len);
7   minfo.tos = inc->i_conn->c_tos;
8   ...
9   // The flag field in the minfo struct is not initialized
10
11  rds_info_copy(iter, *minfo, sizeof(minfo));
12  // The minfo struct is copied to the user-space with the uninitialized 'flag'
      field
```

Listing 5.3: The vulnerable function of CVE-2016-5244. The `rds_info_copy` function copies the `minfo` struct with `flag` field uninitialized.

For spraying the kernel stack using the BPF program as in Listing 5.2, we first checked a leak offset of this vulnerability: the leak offset is *1,568*. We, then, executed the BPF program by calling the `sendmsg()` syscall. However, we could not clobber the offset because the BPF program sprays the kernel stack from offset 1,032 to 1,544 when we use the `sendmsg()` syscall. As we discussed in Section 5.5.3, there are various syscalls that can trigger the `bpf_prog_run()` function and each of them uses a different execution path to the BPF program runner—the stack spraying range is different based on the execution path. In this case, we found that we can clobber the leak offset through the `compat_sendmsg()` syscall by which we can spray the kernel stack from 1,064 to 1,576. Consequently, we could identify the kernel stack base with the guess and check method introduced in Section 5.5.4.

Vulnerability	Leak Size	CVSS	Exploitation Result
CVE-2018-11580	4	2.1	Bypassed KASLR
CVE-2016-4569	4	2.1	Bypassed KASLR
CVE-2016-4486	4	2.1	Kernel stack base
Fixes: 372f525	4	N/A	Bypassed KASLR
CVE-2016-5244	1	5.0	Failed

Table 5.3: Summary of Exploitation Results of Vulnerabilities: We analyzed 4 CVEs and 1 security patch which could not become a CVE entry.

CVE-2016-5244

The CVSS score of this vulnerability is 5.0, which is significantly higher than the other CVEs that we evaluate in this work. Interestingly, we found that this one-byte leak vulnerability *cannot be exploited* through our analysis.

Listing 5.3 shows the vulnerable function where the `minfo` struct with the uninitialized `flag` field is copied to the user-space through the `rds_info_copy` function at Line 22. However, the leak offset of the uninitialized field (1 byte) always becomes `0` before the vulnerable function executes. Therefore, the vulnerability always leaks `0`, even though we can successfully store kernel pointer values at the leak offset.

Summary

In our evaluation, we analyzed four CVEs and one patch in the upstream Linux kernel as summarized in Table 5.3. We showed that our approach can effectively generate exploits. Additionally, the experimental results imply that our community is in need of a more accurate exploitability evaluation system for information leak bugs in Linux kernel so that security implications of bugs can be estimated more correctly.

5.7 Discussion

We discuss about limitations of this work and possible mitigations against stack-based information-leak vulnerabilities.

5.7.1 Limitations

We showed that small leaks can be exploited to identify the KASLR slide (CVE-2018-11580) and the kernel stack base (CVE-2016-4486). Even though our approach to identifying the KASLR slide currently has no limitation in its usage, the BPF-based approach to reveal the stack base cannot be used in the Linux kernel from v.4.14.113 as in Section 5.5.4 (stack spraying is still possible). Therefore, we need a more general method to handle small leaks especially for revealing the stack base. To overcome this limitation, one possible strategy is to analyze the Linux kernel statically to find code gadgets which can modify the kernel stack with user-controlled data. We leave this limitation for future work.

Next, our approach analyzes information-leak vulnerabilities using programs that *can trigger* a vulnerability. Hence, we could not evaluate our approach in a large scale; Instead, we show the effectiveness of our approach against a limited number of vulnerabilities. This is mainly because generating such exploits manually is a time-consuming and complicated task. Even though we know which function has a vulnerability, we should find a proper context and create exploits to trigger it by manually analyzing the kernel source code. To enable large-scale experiments, our approach needs to be incorporated with emerging automatic exploit generation technologies such as FUSE [143].

5.7.2 *Mitigating Uses of Uninitialized Memory*

There are a couple of security features for uninitialized memory uses in the Linux kernel. STACKLEAK clears the kernel stack when syscalls return to the user-space, which was integrated into the Linux kernel upstream from v4.20 [115]. Recently, new configuration options, `CONFIG_INIT_ALL_MEMORY` and `CONFIG_INIT_ALL_STACK`, were introduced to force initialization of stack and heap variables [1]. In addition, many mitigation approaches have been proposed to prevent uninitialized memory uses. Peiró, *et al.* proposed a mechanism for detecting stack-based information-leak bugs of the Linux kernel through static data flow analysis [114]. Garmany, *et al.* have proposed another static data flow analysis framework that finds uninitialized stack memory uses after lifting binaries into an intermediate representation [64]. UniSan is a compiler-based approach to prevent information leaks caused by uninitialized read [96]. UniSan performs byte-level data flow analysis statically for OS kernels and instruments code to initialize data if it leaves kernel without initialization. The kernel memory sanitizer (KMSAN) is a tool to track uninitialized data to check whether the data leaves OS kernels or not, which can be utilized with fuzzers such as the syzkaller [10]. On the other hand, as a runtime defense system for OS kernels, kMVX has been proposed against information-leak vulnerabilities by leveraging the multi-variant execution [112]

5.8 Related work

Exploiting uninitialized memory uses.

Albeit there have been research efforts on controlling uninitialized data to leverage it in other types of vulnerabilities, exploiting stack-based information-leak vulnerability to leak sensitive information such as pointer values pointing to the kernel stack or

kernel code has not been explored yet [61, 97, 147].

Thomas Dullien (also known as Halvar Flake) proposed a search algorithm using call graphs for finding a function that can have a stack frame overlapping with the target memory address [61]. Lu, *et al.* proposed an automated method for writing arbitrary data to uninitialized stack variables through targeted stack spraying [97]. Xu, *et al.* showed common types of uninitialized uses and their potential threats by exploiting two uninitialized use vulnerabilities which can lead attackers to gain arbitrary kernel code executions in the macOS [147].

Automating kernel exploitation.

Automated kernel exploit generation is a demanding task. In addition, even determining the exploitability of bugs requires significant manual efforts. Security researchers have been attempting to address these problems. FUZE [143] proposed to identify useful system calls for kernel use-after-free exploitations by leveraging fuzzing and symbolic execution techniques. KEPLER [142] showed a code-reuse exploit approach that converts a user-provided control-flow hijacking primitives into arbitrary stack overflows, and thus, it bootstraps return-oriented programming (ROP) payload. Chen *et al.* [52] proposed static and dynamic analysis methods to find useful data structures for use-after-free exploitations in the Linux kernel.

5.9 Conclusion

In this work, we proposed a generic approach to exploit uses of uninitialized stack data in Linux kernels to leak pointer values that are pointing to either kernel functions or to the kernel stack. These leaked pointer values can then be used to defeat KASLR and mount future attacks against Linux kernels. Our evaluation results show that we can effectively analyze and exploit stack-based information-leak vulnerabil-

ities through the proposed approach. Our proposed approach exposes the actual exploitability and severity of information disclosure bugs in Linux kernels and will raise awareness of the community on the security impact of these bugs. We expect our findings will help adjust CVSS scoring for information leak bugs inside Linux kernels.

Chapter 6

VIK: PRACTICAL MITIGATION OF TEMPORAL MEMORY SAFETY VIOLATIONS THROUGH OBJECT ID INSPECTION

6.1 Introduction

Temporal memory safety violations, such as use-after-free (UAF) and double-free vulnerabilities, are a critical security issue impacting programs developed in unsafe languages. These violations occur when a program dereferences a dangling pointer which points to a memory location that was allocated and later freed. Memory safety violations may lead to severe vulnerabilities in operating systems and software programs. In 2019, 53% of the UAF vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database lead to remote code execution or privilege escalation [58]. Unfortunately, since many operating systems and user-space programs are still written in memory unsafe languages, memory safety violations remain a clear and present danger to computer security. Therefore, preventing the security impact of temporal memory safety vulnerabilities is of paramount importance, yet it is still an open problem.

Ideally, these vulnerabilities would be found automatically during software development and testing. However, the spatial separation of allocation and deallocation code and the size and complexity of real-world software have kept effective, automated techniques from being developed thus far. As a result, recent research has focused on preventing not the vulnerabilities themselves, *but their exploitation*. These protection techniques span two categories: Static analysis achieve impressive scalability, but lack accuracy because of the inherent difficulty in solving complex program analysis

problems such as inter-procedural data flows and pointer aliasing [143, 134, 57, 89]. As a consequence, most proposed defenses are dynamic, aiming at forbidding unsafe memory reuse [57, 128, 19, 108, 45], validating memory accesses and detecting the use of dangling pointers [106, 129] or preventing the creation of dangling pointers [126, 92, 134, 89, 153, 48] *at runtime*. These solutions show promise in detecting (and defending against) temporal memory violations, but they introduce significant performance and/or memory overhead. For example, the state-of-the-art defenses CRCount [126] and pSweeper [92] incur runtime overhead of roughly 135% and 71%, respectively, on running the `perlbench` program from SPEC 2006. This overhead is inherent to existing dynamic protection mechanism designs: It arises from the use of joint metadata that records relations between pointers and allocated objects [126, 89, 134, 128]

In this work, we propose ViK, a practical runtime defense mechanism against exploits of temporal memory safety vulnerabilities in user-space applications and OS kernels. The core idea backing ViK is *Object ID inspection*: ViK assigns a random ID to every allocated object and stores it in the *unused bits* of the corresponding pointer value (virtual address). While the operating system or the user-space program is executing, ViK inspects the pointer value before each dereference and ensures that the pointer value references the original object for which it was created. ViK also inspects the pointer value before deallocating the object. This design endows ViK with advantages in false positive/negative rates, memory usage, and runtime overhead. First, by storing metadata in pointer values, ViK maintains low memory overhead (averaging 9% on for SPEC CPU 2006). CRCount [126], the most recently published work, shows memory overhead of 55% on the same benchmark [126]. Second, by not relying on data-flow analysis or pointer tracking for identifying the location of pointers, ViK does not have false negatives that stem from type-unsafe pointers or pointer values

temporarily stored in registers or on the stack, whereas existing solutions may introduce such errors and fail to mitigate exploits because of them [18, 126, 57, 134, 89].

Runtime overhead determines the practicality of temporal memory safety defenses. To minimize the runtime overhead caused by pointer inspections, ViK employs a sound static analysis to exclude safe pointer dereferences and only protects potentially unsafe ones. In our experiment on SPEC CPU 2006, ViK exhibits runtime overhead of about 10% on average (similar to userspace-specific state-of-the-art UAF defenses, such as Markus [18], which does not scale to kernel code). Our evaluation also shows that ViK-protected OS kernels have an overall 20% system performance overhead on both Linux kernel and Android kernel. ViK is the first mitigation approach against temporal memory safety violations that scales to modern OS kernels.

The pure-software version of ViK does not rely on hardware features, which makes it applicable to legacy hardware. Nonetheless, emerging hardware features can bring substantial benefit: By employing the Top Byte Ignore (TBI) feature of AArch64 processors, we implemented a hardware-assistant ViK variant (codenamed ViK_{TBI}) that achieves an average full-system runtime overhead of less than 2% when applied on Android kernels. As a result, a major hardware manufacturer is currently deploying ViK_{TBI} in their OS kernels in *actual* smart-automotive consumer devices *in the real world*. To the best of our knowledge, this makes ViK the first kernel-level temporal memory safety system to be deployed in actual real-world products.

6.2 Previous Defenses Against Use-after-free Exploits

Pointer invalidation

Defenses designed to prevent the creation of a dangling pointer either invalidate pointers that point to deallocated memory regions or prevent the deallocation of an object

if there are pointers that point to it [126, 92, 134, 89, 153, 48]. These defenses all maintain additional metadata to track the relationships between pointers and their corresponding objects (memory allocations). Compared to approaches based on safe memory allocation (which will be discussed later), pointer invalidation techniques provide a stronger defense with lower memory overhead. However, pointer invalidation methods usually incur substantial runtime overhead because they have to monitor memory allocations and maintain relevant metadata. In multi-threaded programs, the use of joint metadata could also impose a high-performance overhead because the metadata must be updated in a thread-safe way [126]. Additionally, due to the difficulty (and infeasibility) of performing a sound and complete data flow analysis on programs in general, pointer invalidation defenses inevitably suffer from false negatives. For instance, these defenses do not track the propagation of pointer values through type-unsafe pointers. In addition, they cannot track and invalidate pointers which are stored in registers and on the stack, which can result in further false negatives [134, 89, 153, 48].

Safe memory allocation

Safe memory allocation techniques aim to increase the difficulty of exploiting UAF vulnerabilities. An example is the SLUB allocator (used by the Linux kernel), which allocates small objects to predefined cache slots of certain sizes. The SLUB allocator makes UAF attacks harder as it guarantees that a kernel object only overlaps with a deallocated object with the same size. However, SLUB does not completely mitigate the exploitation of kernel UAF vulnerabilities [145]. There are several similar mitigations designed to prevent the re-allocation of objects to the memory areas previously occupied by a victim object [57, 128, 19, 45, 108]. Although these mitigations make UAF-based attacks even harder than SLUB, they tend to incur high memory

overhead due to the use of new allocation policies (*e.g.*, Oscar allocates each object in a unique virtual memory area [57]). While, MarkUs [18], a user-space memory allocator that delays the reuse of freed memory objects until there are no dangling pointers, showed good memory and performance overhead by using the well-designed metadata and additional threads for inspecting it.

Access validation

Several approaches attempt to prevent UAF attacks by validating every memory access that involves a pointer dereference [106, 129]. While these approaches provide security guarantees and acceptable memory overhead, they all incur substantial runtime overhead because they must check every pointer dereference or update metadata on a regular basis during runtime [106].

6.3 Unused Bits in 64-bit Virtual Addresses

64-bit architectures use 64-bit virtual addresses. However, most modern 64-bit architectures do not fully utilize the 64-bit virtual address space. For example, x86-64, AArch64, RISC-V, MIPS, and OpenSPARC only support virtual addresses up to (or less than) 48 bits, which correspond to a virtual address space of at most 256 TB[1] [75, 20, 76, 103, 131]. We observe that the most significant 16 bits in every pointer value are currently unused for *data pointers* on most processors.

These unused bits have not gone unnoticed. ARM announced pointer authentication instructions in the ARMv8.3-A instruction set, which uses the unused bits in 64-bit pointers to sign and authenticate virtual addresses [117]. To maintain the

[1]On x86-64, the most significant 16 bits of a virtual address (from the 48th bit to the 63rd bit) must be the same as the 47th bit. Otherwise, the processor will raise an exception when accessing the address.

integrity of pointers in specific contexts, developers can generate a pointer authentication code (PAC) and store it in place of the unused bits. This approach is used for detecting (and verifying) changes to a pointer value (i.e., address) rather than validating the relationship between a pointer and a virtual address to which the pointer points. One typical example of pointer authentication is using it to protect the stack pointer. Unfortunately, pointer authentication instructions are *not* able to prevent UAF vulnerabilities directly because UAF can occur regardless of the pointer's authenticity. In recent work, Liljestrand et al. have shown that the pointer authentication scheme is *not able to mitigate temporal safety vulnerabilities* such as UAF [90].

It is worth noting that ViK does not exclude the use of PAC but complements it by providing additional protection on relationships between data pointers and memory objects. It is even possible to use ViK and PAC together on the same pointer: a pointer authentication code can be generated for a pointer containing the object ID issued by ViK in the unused bits. Thus, we can first *authenticate* the pointer value and then *verify* that the pointer value references the original object for which it was created.

In addition, ARM introduced Memory Tagging Extension (MTE) in ARM v8.5 [5] and Application Data Integrity (ADI) is enabled in a number of SPARC processors (M7, M8, S7, T7, and T8) [111]. With MTE, a tag is assigned to each allocated memory region, and only a pointer that has the same tag can access the region. Similarly, ADI utilizes version numbers stored in the unused bits of application's memory pointers and the memory they point to. We expect that MTE and ADI can help prevent memory errors. However, the size of a tag in MTE and a version number in ADI is just 4 bits [9, 83], and thus, can only have 16 possible values. Also, how MTE and ADI can be automatically applied to OS kernels is an open research

question. Currently, ADI can be used for user-space programs on Linux and Oracle Solaris [83, 111], but Linux kernel support for using MTE in user-space programs remains in development [62]. Therefore, it is important to mitigate temporal memory safety violations for OS kernels and for the large set of processors, including the latest Intel chipsets, that do not offer hardware features for detecting temporal memory errors.

6.4 Threat Model

In this work, we focus on mitigating UAF and double-free exploits in *all* programs written in C/C++ that run in either user space or kernels. Similar to existing defenses [18, 126, 92, 57, 134], we focus on heap-related temporal memory safety vulnerabilities as stack-based UAF or double-free vulnerabilities can be handled by use-after-scope or escape analysis [18, 57]. In addition, such stack-base UAF vulnerabilities are rare in reality. Kernel Address Sanitizer (KASan) stopped the support of detecting stack-related UAF errors because the detector is considered "almost entirely useless" [7]. Therefore, it is reasonable to narrow down UAF and double-free vulnerabilities to heap-only.

6.5 Overview

ViK aims to mitigate temporal memory safety vulnerabilities by detecting and preventing the use of dangling pointers at runtime. As illustrated in Figure 6.1, ViK adds a middle layer between the compiling stage and the linking stage: ViK takes as input a program compiled to LLVM IR, identifies all allocation and pointer dereferencing sites, and performs *static instrumentation* on the LLVM bitcode to insert object-ID-specific logic. Specifically, ViK replaces each memory allocator with a new one ($alloc_{vik}(x)$) and adds an object-ID check ($inspect(p)$) at (1) pointer dereferencing

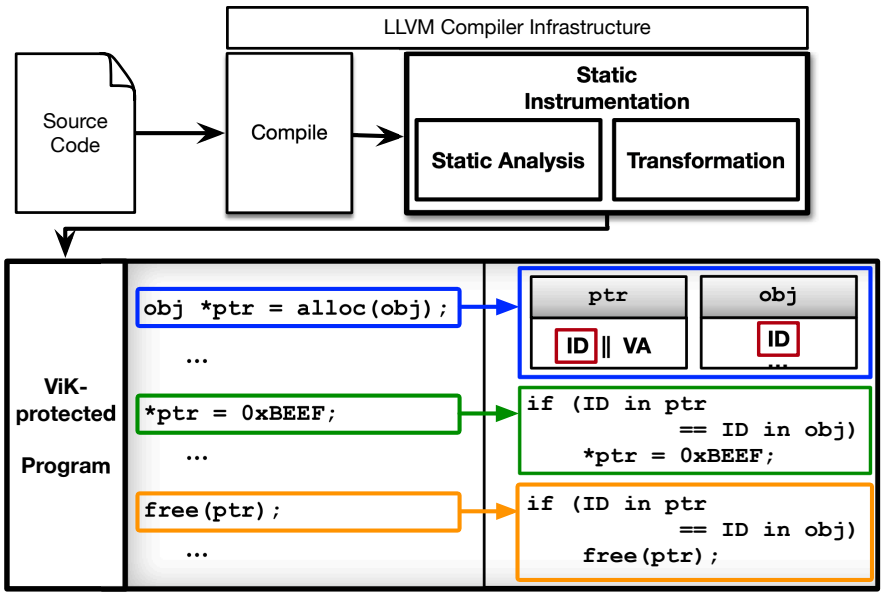


Figure 6.1: Overview of the Static Instrumentation for Applying ViK and a ViK-protected Program: The instrumentation process first takes LLVM IR as input. Then, it performs static analysis to replace allocators and insert object ID inspection code before pointer operations. At runtime, each object ID is copied to the unused bits of the pointer and the base address of the newly allocated object. Before any protected pointer is dereferenced, ViK inspects that the object ID stored in the pointer matches the one stored in the object.

sites and (2) when an object is deallocated as defined in Section 6.7.

The core of ViK consists of the following three main steps.

I. ViK assigns a random object ID to an object when one is allocated in the heap memory.

II. ViK copies the assigned object ID to the unused bits of a pointer value and to a reserved field at the base of the newly allocated object. This way, ViK creates a correspondence between the pointer value and the object.

III. At runtime, when the pointer value is dereferenced, ViK inspects the object ID and allows the dereference only when the ID stored in the pointer value matches the one in the object. Also, ViK inspects the object ID when an object is freed to prevent the double-free.

Dynamic instrumentation in ViK

To maximize compatibility with legacy programs and systems, ViK is designed to be independent of hardware support on 64-bit architectures. By embedding object IDs directly into pointer values, ViK avoids two common dependencies, relied on by other defenses, that cause excessive runtime overhead, memory overhead, and concurrency challenges: the use of in-memory central metadata and the tracking of pointer propagation. Because it is stored in the pointer itself, object IDs used by ViK always move with the pointer value to which they belong whether the pointer value is loaded to a register, propagated into other pointers, or spilled onto the stack. However, a pointer value might not point to the base address of an object. To find the base address of any object pointed by a protected pointer value, ViK additionally embeds a *base identifier* into each object ID, which will be discussed, together with the design of object IDs, in Section 6.6.

Static optimization in ViK

Our design allows it to scale to OS kernels. To accomplish this, ViK must be able to defend programs that have extreme numbers of pointer accesses (*e.g.*, there are about 2.3 million pointer dereferences in Linux kernel 4.12), and naïvely inspecting every memory access will incur an impractical (or, minimally, unnecessary) runtime overhead. Therefore, we aim to minimize the number of pointers to inspect by limiting the dynamic instrumentation only to those dereferencing sites that are potentially

unsafe. To this end, ViK conducts an inter-procedural static data-flow analysis to identify memory accesses that are considered to be safe from UAF exploits and exclude them from the inspection.

One result of this optimization is that ViK omits object ID inspection on dereferences of pointers that are never stored in global regions or the heap, deeming them "UAF-safe." Because these temporal pointer values only exist on the stack, have a very short lifetime, and are generally accessed by a limited amount of code, their use as a dangling pointer in a UAF exploit is extremely unlikely.[2] Therefore, ViK does not inspect pointer values that are only stored on the stack and are never copied to the heap or into global variables. Prior work shares this trade-off: For example, DangNull only tracks pointers located on the heap, which is more aggressive than our assumption [89]. Our protection model covers more dereference sites than DangNull [89] and the same amount of dereference sites as CRCount [126] and pSweeper [92]. Additionally, ViK covers pointer values in registers, weak-typed pointers, and pointers that are spilled onto the stack, which are not covered by prior work. We will extensively discuss this in Section 6.7.

Mis-detections

ViK's design guarantees an absence of false positives (mistaken UAF detection when no UAF is taking place), but false negatives (no UAF detection when a UAF is taking place) can occur if two objects are assigned the same object ID or if an object assumed to be UAF-safe is actually attackable. For the former case, *object ID collisions* occur with a very low probability. We believe that there is sufficient entropy in the object

[2]Note that, if a heap-stored dangling pointer is used to attack the object pointed to these UAF-safe pointers, ViK will still catch the attack.

IDs to defeat an attacker's attempt of circumvention[3] as we will discuss in Section 6.6.2 and Section 6.9.3, respectively. The latter case is also rare. In fact, the only example that is known to us of a UAF-safe object being used to trigger a UAF involves another necessary step that ViK *does* catch, which we term *delayed mitigation* and will discuss in Section 6.9.3.

6.6 Object ID

ViK generates an object ID for each object and embeds the ID into both the pointer value and the object. We first present how ViK securely generates object IDs (Section 6.6.1) and then further discuss object ID entropy (Section 6.6.2).

6.6.1 Generating Object IDs

As depicted in Figure 6.2, an object ID has 16-bits and is comprised of two parts of variable lengths: an *identification code* and a *base identifier*.

Identification code

The identification code is a random number generated by the ViK allocator (defined in Section 6.7). ViK uses the identification code to uniquely identify each allocated object. The size of the identification code changes depending on the size of the base identifier.

Base identifier

A pointer value does not necessarily point to the base address of an object; instead, it may point to any field inside the object. We need to be able to map a pointer value

[3]Especially for OS kernels, consider that each incorrect ID guess by the attacker will crash the entire system.

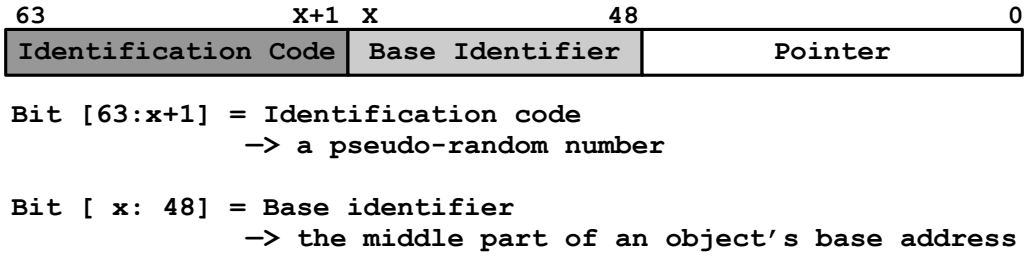


Figure 6.2: The Object ID Consists of Two Parts: (1) identification code and (2) base identifier.

```
1 get_base_identifier(pointer, M, N) {
2   Bwe = (pointer & (2^M − 1)) >> (N);
3   return BI;
4 }
5 get_base_address(pointer, M, N, BI) {
6   BA = (pointer & ∼(2^M − 1)) | (Bwe << N);
7   return BA;
8 }
```

Listing 6.1: Pseudo code for extracting the base identifier and recovering the base address of an object from a pointer value. Both operations only use bitwise instructions.

to the base address of the object to which it points so that ViK can find the object ID, which is stored as the first member of the object. Therefore, ViK introduces *base identifier*s, which are used to find the base address of any object.

ViK aligns allocated objects to a predefined alignment of 2^X bytes, which essentially creates *slots* of at least 2^X bytes. By aligning the addresses of objects, the least significant X bits of all objects' base addresses must be zero. An object may require one or more slots depending on its size. For example, if a slot is 16 bytes, a 24-byte object will use two slots.

ViK uses two predefined constants M and N to determine the alignment (which

is also the size of each slot). 2^M is the maximum size (in bytes) of objects that can be covered by using slots of size 2^N (bytes) with a base identifier of $M - N$ bits. For example, suppose M and N are 12 and 6, respectively; the maximum size of any object is $2^{12} = 4096$ bytes; and the size of each slot is $2^6 = 64$ bytes. The base identifier will be $12 - 6 = 6$ bits long.

Once the constants are determined, the base identifier can be calculated from the start address of an allocated memory region as shown by Lines 1–4 of Listing 6.1. During runtime, ViK can recover the base address of an object from any pointer value, as in Lines 5–8 of Listing 6.1. ViK only uses bitwise operations to find base addresses of objects. It does not need memory accesses, which helps keep runtime overhead low.

Determining the constants

M and N must be configured before ViK's instrumentation. Since slots are the smallest allocation unit in ViK, using large slots can cause excessive memory overhead. ViK asks the user to specify M and N with the assistance of the knowledge of object sizes. ViK helps users to determine optimal choices of the two parameters by identifying sizes of all the involved objects in the target program, which is straightforward to do with a compiler pass. Note that determining the optimal M and N is a one-time effort for each target program. We will demonstrate the process of determining M and N on Linux kernel.

Since ViK can potentially support multiple sets of M and N to generate base identifiers for various sizes of objects, we can minimize the memory overhead by using different constant values for different types of objects. During the instrumentation phase, base identifier calculation functions with different constants can be injected for each pointer operation, depending on the size of each object and predefined constants

for it.

6.6.2 Entropy of Object IDs

The effective entropy of an object ID is equal to the size of the identification code. That is, the base identifier does not add any security. When a system is attacked, an object can be allocated at the exact same address where the victim object was placed, and this newly allocated object would have the same base identifier as in the victim object. This entropy may seem low, but we believe it is sufficient to stop attacks. In our evaluation, we used 10-bit identification codes (which have a collision ratio of about 0.09%), which is equal to the entropy of the Linux kernel's Kernel Address Space Layout Randomization (KASLR) [81], and ViK could successfully defeat all attacks that use known vulnerabilities in Linux kernel without any collision of object IDs. To make a UAF attack successful, an attacker must re-allocate an object that has the same base identifier as the victim object, and the newly allocated object must have the same object ID that is randomly chosen (the random space does not decrease by allocating new objects). Also, there is only one chance to launch an attack: The kernel will panic upon failed attacks due to an invalid memory access via a pointer value returned from the `inspect()` function (Definition 3). Therefore, even though the 0.09% collision ratio may not seem very low, it will still be difficult for an attacker to bypass the mitigation in practice.

6.7 Instrumentation

In this section, we elaborate on ViK's core logic—the instrumentation procedure. ViK instruments the target program at the LLVM IR level through a two-step process. First, ViK runs a static data-flow analysis to determine all optimal locations for adding pointer inspection logic (we scope pointers to protect in Section 6.7.1 and detail

the static analysis in Section 6.7.2). Second, ViK instruments the LLVM bytecode of the target program to insert the pointer-inspection logic (discussed in Section 6.7.3).

Common Terms

Before presenting the design details of ViK, we define terms that we will use throughout this work: the memory allocator and the inspect function of ViK.

Definition 2. *ViK memory allocator ($alloc_{vik}()$) allocates a specific size (x) of heap memory and returns its start address (p) along with an object ID (id). The ID is also stored in the allocated memory.*

$$alloc_{vik}(x) \rightarrow p_{id},\ id \in [p...(p+x)] \in \{mem_{heap}\}$$

p is the start address (a virtual address in the 64-bit canonical form), id is an assigned object ID, and p_{id} is the combined form that contains id in p.

Definition 3. *The inspect function ($inspect(p)$) returns p if and only if the id in p corresponds with the id in a memory object to which p points.*

$$inspect(p_{id}) \rightarrow p \Leftrightarrow id \equiv p \in \{mem_{heap}\} \wedge id \in [p...(p+x)]$$

Therefore, in ViK-protected programs, UAF attacks are mitigated because one of the following two cases will hold for dangling pointers: (1) it will have an object ID that is different from the ID of the object to which it points; or (2) it will not point to a valid memory region on the heap.

6.7.1 UAF-safe Pointers

Because ViK defends against UAF attacks by validating memory accesses, its runtime overhead is proportional to the number of pointer inspection sites that ViK

inserts into the target program. Therefore, our optimization goal is to minimize the number of inspect functions in the protected program.

Based on our analysis of prior UAF exploitations in the Linux kernel and user-space applications, we assume that pointer values pointing to global or stack variables are unlikely to be used to exploit corresponding UAF vulnerabilities [7, 134, 89]. This is because the variables such pointers point to can never be programmer deallocated (freed) and have a very short lifetime. Therefore, we deem memory accesses using these pointer values as very likely to be safe from UAF attacks, *i.e.*, *UAF-safe*. It is worth noting that state-of-the-art UAF defenses make the same assumption [7, 126, 143, 57, 134, 89]. Also, the Kernel Address Sanitizer (KASan) deprecated a function for detecting UAF errors related with stack-located objects because they did not find such bugs for two years of fuzzing effort [7]. A pointer value of a pointer operation is regarded as UAF-safe if all memory accesses through this pointer value are UAF-safe before the dereferencing site. To reduce runtime overhead, ViK finds all UAF-safe pointer values using static analysis in a principled manner and excludes them when inserting the pointer-inspection logic.

In addition, we consider pointer values that have *never* been stored in the heap or global variable to be UAF-safe. ViK, thus, does not inspect pointer values stored only on the stack, before they are copied to the heap or global variable. If a pointer value of a newly allocated object stays only on the stack, it can only be used within the current context, and thus, a UAF bug can happen only when the object is freed and the pointer value is used after the deallocation in the same context. To exploit this kind of UAF bugs, an attacker must allocate objects to overlap with the victim object after it is freed and before the dangling pointer is used. Also, the attacker cannot execute code in the same context where a UAF bug occurs. Consequently, in single-threaded programs, exploiting such UAF bugs is not possible. In multi-

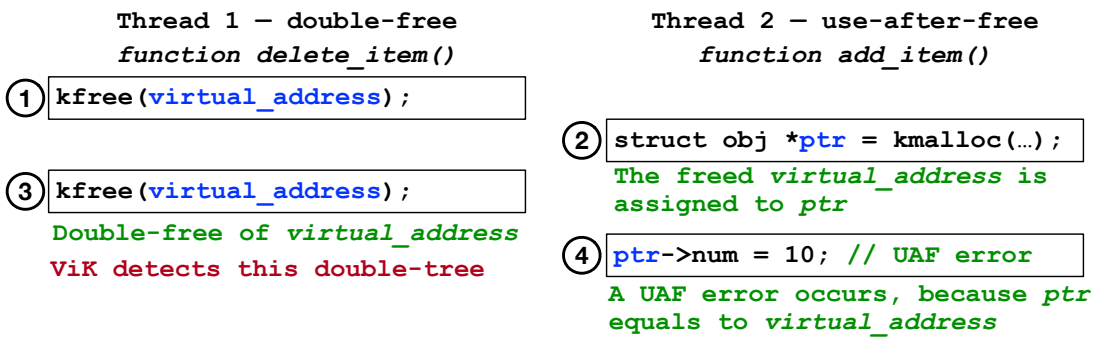


Figure 6.3: An Example of UAF Error Based on a Double-free Bug: In Thread 2, a UAF error occurs with a pointer value stored only on the stack by the double-free bug at ③ in Thread 1. This type of UAF error can be exploited if an attacker can allocate a new object between ③ and ④. ViK inspects the object ID when an object is deallocated at ③ and stops the attack.

threaded programs such as OS kernels, it is extremely difficult to overlap the victim object within the limited time frame because we cannot know when the victim object is freed and accessed again. The only possible UAF exploitation case currently known, even with a pointer value that has never been stored in the heap of a global variable, is leveraging a double-free bug which can lead to a UAF error as illustrated in Figure 6.3. However, ViK can prevent the double-free because it inspects the pointer ID when an object is deallocated. Hence, all memory accesses using such pointer values are considered to be UAF-safe *until* such pointer values are copied to global variables or the heap. Any pointer value copied from the heap or global variables is always considered to be *UAF-unsafe*.

Definitions of UAF-safety

For the sake of clarity, we define UAF-safe pointer values as follows, which holds through the entire static analysis.

Definition 4. *Any pointer value that points to a global or a stack variable is UAF-safe. Also, a pointer value that points to the heap is UAF-safe if and only if the pointer value has never been stored in the heap or a global variable.*

The above definition states that only globally known pointer values pointing to the heap are relevant to *exploiting* UAF vulnerabilities.

Definition 5. *Assuming that a function takes a pointer value as one of its arguments, this pointer value is UAF-safe if and only if the pointer value is UAF-safe in the caller function.*

When only UAF-safe pointer values are used as arguments of function calls, these arguments are UAF-safe in the callee function since they are still used as stack variables and may not be accessed by other threads unless they are copied to a global variable or the heap.

Definition 6. *Assuming that a function returns a pointer value, the left-hand-side (lhs) pointer value at the call site in the caller function is UAF-safe if and only if the pointer value is UAF-safe in the callee function.*

If the returned pointer value is UAF-safe at the return site of the callee function, the *lhs* pointer value at the call site will be a local variable in the caller function before it is copied to a global variable or the heap. Hence, this pointer value is still UAF-safe before the copying happens. However, if we do not know whether or not the returned pointer value is UAF-safe in the callee function, for soundness, we must make an under-approximation by assuming that the pointer value in the caller function is *UAF-unsafe* from when the pointer value is returned. This can happen when our analysis does not consider the callee function or cannot determine the UAF-safety of a pointer value returned by the callee function. Likewise, any pointer values that are not UAF-safe are considered to be UAF-unsafe for soundness.

6.7.2 Static Analysis

The goal of ViK's static analysis is to determine if a pointer value is UAF-safe at an accessing site of this pointer value. For accuracy, ViK's static analysis is both *flow-sensitive* and *path-sensitive*. In intra-procedural analysis, since the UAF-safety of a pointer value may change depending on its execution path, simple backtracking on the data flow is insufficient to determine the UAF-safety of a pointer value at an arbitrary pointer operation. To consider all possible cases that affect the UAF-safety of a pointer value, our static analyzer contains a Reaching Definition Analyzer (RDA) that works on LLVM bitcode instructions. The RDA recovers all possible data flows that can reach a pointer operation. We regard a pointer value as UAF-safe if and only if all its uses are UAF-safe according to Definitions 4, 5, and 6.

Step 1: Intra-procedural Analysis

We conduct an intra-procedural analysis on each function, to find UAF-safe pointer values, by analyzing all pointer operations in a function. To this end, we execute the RDA for every pointer operation. According to Definition 4, a pointer value is UAF-safe if it points to a stack variable or a global variable. In such cases, ViK does not place *inspect*() functions for pointer operations of that pointer value. Pointer values copied from the heap or global variables must be inspected by ViK since they are considered UAF-unsafe.

Next, we mark a pointer value as UAF-unsafe if the pointer variable may hold a return value from another function. As the first step is intra-procedural, we do not know if the returned pointer value is UAF-safe or not. However, we mark pointer values with return values returned from basic allocators (*e.g.*, *malloc*() in libc or *kmalloc*() in the Linux kernel) as UAF-safe, since the pointer values returned by

basic allocators are obviously UAF-safe. Lastly, if a function argument is a pointer value, we deem this pointer value UAF-unsafe. The UAF-safety of these pointers can be updated after finding UAF-safe arguments and return values in future steps.

Step 2: Analyzing UAF-safe heap addresses

Once the initial intra-procedural analysis is complete, we build a call graph for each module (a source file or an object file after compiling) of the target program. The call graph is used to determine the UAF-safety of pointer values in our inter-procedural analysis.

First, we analyze pointer values that hold the pointer values returned from basic allocators. Note that immediately after a basic allocator returns, the pointer value (which is the address of the newly allocated object) is unknown to other threads. If there is an instruction that copies a pointer value to a global variable or the heap, the pointer value must become UAF-unsafe *after* the execution of the `store` instruction. Tracking data flows related to these pointer values is straightforward since they are local variables located within the stack frame.

Step 3: Finding UAF-safe function arguments

To find UAF-safe arguments for every function call, we visit functions from the dominator node based on the call graph. We check every call instruction and its arguments. If a pointer is used as an argument, we decide whether or not the argument is UAF-safe. We add an attribute to the argument to mark it, only if an argument is UAF-safe for every case before the call instruction. We repeat this process until we have visited all functions in the call graph. Because we limit the range of our static analysis to a single module, we omit checking arguments of functions that go beyond the analysis range.

If an argument of a function has been marked as UAF-safe, we visit the function to run RDA again to change the UAF-safety of pointers in each pointer operation that we have visited before.

Step 4: Finding UAF-safe function return values

We find UAF-safe return values from the post-dominator nodes in the call graph. Similar to the process of finding safe arguments, we visit functions that return a pointer value and analyze their return values. In a function, it is possible to have multiple return instructions depending on execution paths. Therefore, we perform a path- and flow-sensitive analysis for each return instruction and decide their UAF-safety. Only if all possible return values are UAF-safe, do we mark the pointer value as a UAF-safe.

As in Step 3, it is necessary to execute the RDA again for every function after finding UAF-safe return values. The UAF-safety of pointer values can be changed depending on the UAF-safe return values.

Step 5: Finding the first memory access in a function

Through the previous analyses, the UAF-safety of all pointer values has been decided. In the last step, we optimize the number of *inspect*() functions for each function. The idea of this step is to inspect only the very first pointer operation of a UAF-unsafe pointer value in a function, and thus, UAF-unsafe pointer values are inspected once in every function. We carefully take this step using RDA to detect changes of pointer values along all possible execution paths. We thus are able to find the first memory access among pointer operations using the same pointer value. This optimization significantly decreases the runtime overhead, but it may lead to false negatives. We discuss its security implications further in Section 6.9.3.

```
1  struct obj *global_ptr = NULL;
2  void add(struct obj *ptr) { *ptr += 5; /* safe */ }
3  void sub(struct obj *ptr) { *ptr -= 5; /* unsafe -> inspect() */ }
4  void make_global(struct obj *ptr) { global_ptr = ptr; }
5  void ptr_ops(struct obj *arg) {
6    struct obj *safe_ptr = malloc(4); /* safe */
7    struct obj *unsafe_ptr = get_obj(); /* unsafe */
8    *safe_ptr = 10; /* safe */
9    *unsafe_ptr = 10; /* unsafe -> inspect() */
10   add(safe_ptr);
11   sub(unsafe_ptr);
12   if( arg == 0 ) { make_global(safe_ptr); /* safe -> unsafe */ }
13   else {
14     *safe_ptr = 10; /* safe */
15     global_ptr = malloc(4);
16   }
17   *safe_ptr = 0; /* unsafe -> inspect() */
18   *unsafe_ptr = 0; /* unsafe -> restore() */
19   ...
```

Listing 6.2: An example of the static analysis result. Our flow-sensitive and path-sensitive static analysis helps ViK reduce the performance overhead significantly as well as provide robust security guarantees against UAF attacks.

Running example

Listing 6.2 shows an example of our static analysis result. In the `ptr_ops` function, `unsafe_ptr` has a UAF-unsafe return value from the `get_obj` function, and thus, is unsafe even though the pointer is a stack variable. Also, the `arg` is a pointer value whose UAF-safety we do not know. Therefore, these pointer values' operations should be inspected by ViK. `safe_ptr` is a safe pointer value (because the pointer value has not been stored in the heap or a global variable) and its operations are not inspected until the point where it may turn into an unsafe pointer value (from Line 23). However, the result of the function call at Line 23 does not affect the

UAF-safety of the `safe_ptr`'s pointer operation at Line 26 since the code is under the `else` condition. Hence, ViK does not inspect the pointer operation on Line 26; however, ViK must inspect the pointer operation on Line 30. ViK omits the inspection of `unsafe_ptr`'s pointer operation on Line 31 because the unsafe pointer value has already been inspected by ViK *and* it does not contain a new unsafe pointer value copied from somewhere. If a new unsafe pointer value might be in `unsafe_ptr`, ViK will inspect the pointer operation.

Our static analysis can confirm that an argument of the `add` function is a safe pointer value whenever the function is called; thus, ViK does not inspect the pointer operation using the argument on Line 4. On the contrary, `unsafe_ptr` is used as an argument of the `sub` function. Hence, the pointer operation inside the `sub` function must be inspected, even though the pointer value has been inspected before the function is called.

As shown in the example, our static analysis is flow-sensitive and path-sensitive, which not only significantly reduces the performance overhead, but also helps provide robust security guarantees against UAF attacks.

6.7.3 Transformation

In the transformation phase, the LLVM instrumentation pass first inserts the *inspect*() function before pointer operations that ViK must inspect during runtime. In the *inspect*() function, ViK does not store the restored virtual address back to memory, but stores it only in a register temporarily and accesses the heap by referencing the register. We thus need to insert a *restore*() function before all the other pointer operations using ViK-protected pointer values. The *restore*() function is used to temporarily recover the canonical form of a virtual address through only a bitwise operation. Consequently, ViK-protected pointer values must go through either the

inspect() or *restore*() function before a pointer operation. When deallocating an object, only the *inspect*() function is used. ViK inlines these functions at each inspection site. Compared to inserting a *call* instruction that invokes the inspection function, inlining increases the size of protected programs, but it is critical to lowering the runtime overhead.

Next, ViK replaces function calls to basic allocators and deallocators with call instructions that will invoke the wrapper functions. When a new object is allocated, the wrapper functions will generate an object ID, store the object ID at the base address of the object, and return it to the caller as part of the pointer value. In wrappers for deallcaotors, ViK inspects the object ID before an object is deallocated.

Pointer arithmetic

Since ViK only uses the unused bits of a pointer value, all legal pointer arithmetic operations ("`+`", "`++`", "`-`", and "`--`") can be used on ViK-protected pointers without restoring them first. In rare cases, pointer values may be used in comparison (e.g., `ptr1 == ptr2`), where the object IDs in the two pointer values are different if they are not derived from the same base pointer. In such cases, ViK will restore the pointer values before comparing pointers.

6.8 Implementation

We implement ViK in both OS kernels (Linux kernel 4.12 and Android kernel 4.14) and user-space applications through an LLVM pass for static instrumentation. The static instrumentation pass is implemented in LLVM v4.0.

6.8.1 Kernel Implementation

We show evaluation results using two ViK-protected kernels on widely used and long-term supported architectures (x86-64 and AArch64). Because instrumentation is done at the LLVM IR level, ViK is fully architecture-agnostic for user-space protection and mostly architecture-agnostic for kernel protection, as long as unused pointer bits are available on the target architecture. The only step to adapt ViK to new architectures, which is only needed done for kernel protection, is to patch a small amount of inline assembly code, such as inline functions implemented in `atomic.h`, because LLVM IR passes cannot analyze, modify, or rewrite assembly code. We added or modified 446 and 363 lines of code for Linux and Android kernels, respectively. The Linux kernel has many basic memory allocators, and our implementation handles all allocators of the `kmalloc` and `kmem_cache_alloc` family. Note that we excluded source code related to the booting process from instrumentation because these functions will no longer execute after booting is complete.

Inspection logic

Since the *inspect*() function checks every pointer dereference, its implementation is critical to minimizing ViK's runtime overhead. Therefore, we implemented the *inspect*() function in a conditional-instruction-free manner. We only use bitwise instructions to inspect pointer values, however, the *inspect*() function must still raise an exception when a pointer value and an object have different object IDs. The key idea is outsourcing the job of raising exceptions upon unmatched object IDs to the CPU.

Listing 6.3 shows the pseudocode of *inspect*() function that consists of bitwise instructions and one memory access for loading the object ID from the heap. First,

```
1 void inspect(pointer_value) {
2   PTR_ID = pointer_value >> 48;
3   BI = PTR_ID & 0x003f;
4   BA = (pointer_value & ∼(2^M − 1))|(BI << N)|0xffff000000000000;
5   OBJ_ID = *BA;
6   pointer_value = pointer_value & (∼(PTR_ID ^ OBJ_ID) << 48);
7   //If two object IDs do not match, dereferencing this new pointer_value will
      raise an exception
8 }
```

Listing 6.3: The pseudocode of the *inspect*() function which only consists of bitwise operations and a load operation for minimal execution overhead. Protected pointer values are restored to their canonical forms if object IDs are matched.

the function extracts the object ID from a pointer value by bit shifting (Line 3). Second, on Lines 4 and 5, it obtains the base identifier and the base address of the object. Third, it loads the object ID from the actual object stored in memory. Then, the *inspect*() function performs a bitwise `XOR` operation using object IDs stored in the pointer value and the object. The result of this operation is used for a bitwise `AND` operation with the pointer value. If the two object IDs are identical, the pointer value will be of the canonical form (for the kernel, all the unused bits will be 1), and thus, the pointer value will be properly dereferenced and the corresponding object will be accessed. Otherwise, the processor will raise an exception because the pointer value is not in the canonical form.

Enforcing memory alignment

In ViK-protected programs, all memory objects must be located at aligned memory addresses that are derived from the constant N, which is used for generating the base identifier. However, the basic allocators of the Linux kernel do not guarantee this

special memory alignment requirement. Therefore, we enforce memory alignments by wrapping basic allocators in custom wrapper functions in which additional bytes are added to enforce alignment.

The wrappers execute the following operations: (1) When an object is allocated, they allocate $(2^N + 8)$ bytes more than the size of the object, where 2^N bytes is the size of a basic alignment unit. The additional 8 bytes are used for storing an object ID. (2) The wrappers then determine a base address that is aligned to 2^N within the allocated memory region. Because the wrapper allocated an additional 2^N bytes, there must be an address that is aligned by 2^N bytes. (3) The wrappers store the object ID at the base address. (4) The wrappers return a pointer value with a value of the base address plus 8, after storing the object ID into the pointer value's unused bits. If the aligned memory address is $X + 2^N$ (where X is a virtual address returned from the basic allocator), the wrappers store the object ID at the address $X + 2^N$ and the object will use memory from address $X + 2^N + 8$. Because the wrappers allocate an additional $2^N + 8$ bytes, the object can be stored from the virtual address $(X + 2^N + 8)$.

6.8.2 ViK$_{TBI}$ for AArch64 on Android kernel

As a selective implementation feature, we present ViK$_{TBI}$ using the Top Byte Ignore (TBI) feature of recent ARM processors, which achieves much lower performance overhead. With TBI, software can use the most significant 8 bits of the virtual address to hold additional information about an address. By employing this hardware feature, ViK can utilize the 8 bits without handling it in software. However, because only 8 bits are available, we do not use the base identifier in ViK$_{TBI}$ in order to have 8-bit entropy for object IDs. This implies that only pointer values that point to the base addresses of objects can be inspected. Also, when an object is created, we insert

padding bytes and store an object ID right before the base address of an object so that the ID can be accessed via the base address. Albeit ViK$_{TBI}$ cannot provide as strong of security as ViK can, its high efficiency makes it very practical. We evaluate and discuss the security effectiveness and performance overhead in Section 6.9. A major hardware vendor is currently deploying ViK$_{TBI}$ implementation in the OS kernels of smart automotive consumer devices, which shows ViK's practicality.

6.8.3 User-space Implementation

We also implement ViK for C and C++ user-space programs. The user-space version of ViK is the same as the kernel-space ViK except for the following aspects:

- User-space programs use different allocators than the kernel, so the instrumentation pass creates appropriate wrappers for memory allocations such as malloc and calloc.
- In user space, valid pointer values have the first 16 bits as 0, instead of 1 in the kernel. The *inspect*() function is changed in user-space ViK to account for this difference.
- User-space programs may use shared libraries that are ViK-unaware. ViK can be used in programs with ViK-unawre libraries. However, similar to the other compiler-based approaches [126, 134, 89], if a library is not instrumented by the static instrumentation pass, pointer values that come from the ViK-unaware shared library cannot be inspected in ViK-protected programs.

6.8.4 Determining Constants

ViK requires two predefined parameters M and N (see Section 6.6.1) for object ID generation. Our proof-of-concept implementation of the instrumentation pass has a functionality to provide the sizes of dynamically allocated memory objects so that we

Allocation size (byte)			M	N	$M-N$	Alignment	Percentage
	x	< 256	8	4	4	16	76.73%
$256 \leq$	x	< 4096	12	6	6	64	21.31%

Table 6.1: The Sizes of Structures Dynamically Allocated in Linux Kernel 4.12: Roughly 98% of structures is smaller than 4 KB.

can analyze them and decide the two parameters. Table 6.1 shows our results on the size of objects dynamically allocated in Linux kernel 4.12. This object memory size analysis helps with ViK's effectiveness and is done one-time for the kernel. Based on this analysis, we found that over 98% of the kernel memory objects are smaller than 4 KB. In the Android kernel, about 98% of memory objects are smaller than 4 KB as well. Therefore, for the security evaluation, we used 6-bit base identifiers with the parameters M=12 and N=6 to have 10-bit identification codes for all objects, and we did not assign an object ID for the objects which are larger than 4 KB. We set the constants according to the M and N shown in Table 6.1 for evaluating the memory overhead of ViK. It is worth noting that different sets of the constants (M and N) can be used for optimal memory overhead on each system. The current prototype of ViK only supports this set of parameters, and we leave this implementation improvement as future work.

6.9 Evaluation

In this section, we evaluate the effectiveness and performance of ViK in protecting OS kernels and user-space programs from UAF and double-free attacks. We first state the experiment setup (Section 6.9.1) and show the results of kernel instrumentation (6.9.2). We then evaluate the effectiveness of ViK (Section 6.9.3) as well as its runtime and memory overhead on both OS kernels (Section 6.9.4) and user-space programs

(Section 6.9.5).

6.9.1 Experiment Setup

We evaluate ViK on both user-space programs and OS kernels. For user-space applications, we compiled a subset of C programs from SPEC CPU 2006 by ViK-enabled LLVM. For OS kernels, we built Linux kernel 4.12 on x86-64 and Android kernel 4.14 on AArch64. For the Android kernel, we evaluated ViK on a development board featuring an ARM Cortex-A76. All other experiments were conducted on a workstation with an Intel i7-6700 CPU with Ubuntu 18.04 x86-64.

Optimization modes

We evaluate ViK-protected OS kernels in the following optimization modes.

ViK$_S$: An *inspect*() function is inserted for every dereference of possibly UAF-unsafe pointers. This mode is expected to be slower than others.

ViK$_O$: All optimization methods presented in Section 6.7.2 are enabled. In this mode, ViK only inspects the first object access of each UAF-unsafe pointer in every function.

ViK$_{TBI}$: ViK is implemented using the Top Byte Ignore (TBI) feature of AArch64, where ViK inspects only pointer values that point to the base address of objects.

6.9.2 Kernel Instrumentation Results

We measured the number of inserted inspection functions (*inspect*()) and the increase of image sizes after deploying each of the three variants of ViK on Ubuntu and Android kernels in different architectures. Our results, in Table 6.2, show that in both kernels, ViK$_S$ inserted inspection functions for around 17% of pointer operations,

Kernel & Architecture	Mode	Image size (MB)		Build time		# of pointer operations	# of *inspect*() functions (%)
		Original	ViK (increased)	Original	ViK (increased)		
Linux kernel 4.12 x86-64	ViK$_S$	36.26	63.38 (27.12%)	19m 11s	26m 31s (7m 20s)	2,401,337	421,406 (17.54%)
	ViK$_O$		48.33 (12.07%)		22m 10s (2m 59s)		91,134 (3.79%)
Android kernel 4.14 AArch64	ViK$_S$	189.09	208.20 (19.11%)	23m 3s	27m 32s (4m 29s)	2,012,421	333,020 (16.54%)
	ViK$_O$		200.94 (11.85%)		25m 38s (2m 35s)		78,782 (3.91%)
	ViK$_{TBI}$		200.93 (11.84%)		25m 30s (2m 27s)		25,969 (1.29%)

Table 6.2: Statistics of ViK-protected Linux Kernel 4.12 (x86-64) and Android Kernel 4.14 (AArch64): About 17% of all pointer operations involve UAF-unsafe pointers. ViK$_O$ and ViK$_{TBI}$ instrument much fewer pointer operations than ViK$_S$ does.

which means the static analysis regarded 17% of all pointer operations as potentially operating on UAF-unsafe pointers. The other 83% of pointer operations were UAF-safe and do not need any protection. Moreover, in ViK$_O$ mode, our results show that only 4% of pointer operations must be inspected by ViK: ViK$_O$ decreases runtime overhead by omitting protection for over three quarters of all UAF-unsafe pointer operations. ViK$_{TBI}$ further reduces runtime overhead by only instrumenting less than 8% of pointer operations that are protected by ViK$_O$.

6.9.3 Security Effectiveness

ViK aims to mitigate UAF exploits with *no* false positives (i.e., incorrectly blocking pointer accesses that are not UAF), which is guaranteed by the design of ViK. Nonetheless, ViK may have false negatives and allows a UAF exploit: If an object that has been re-allocated to the freed region has the same object ID as the victim object because of an object ID collision, ViK cannot mitigate the UAF exploit. Fortunately, as discussed in Section 6.6.2, the probability of an object ID collision is small enough to make ViK practical.

Another case of false negatives occurs only in ViK$_O$ where ViK does not mitigate the exploit immediately but instead exhibits a *delayed* mitigation. Figure 6.4 shows

```
1  void race() {
2      global_ptr->num = 1;
3      ...
4      global_ptr->num = 0;
5      ...
```

```
6  void dealloc() {
7      free(global_ptr);
8  }
9
10
```

Figure 6.4: Example Code Snippets That Can Cause a UAF Error via a Race Condition.

a case where a UAF exploit occurs due to a race condition. If `dealloc()` is executed at any time before the last dereference of `global_ptr`, this variable will become a dangling pointer and will cause a UAF. ViK$_S$ inspects every UAF-unsafe pointer operation (at both Line 2 and Line 4) and mitigates the UAF exploit. However, ViK$_O$ only inserts inspection functions at Line 2 of `race()`. It is then possible for a UAF to occur if the object that `global_ptr` points to is deallocated between executing Line 2 and Line 4. In the worst-case scenario, the attacker frees the victim object between Line 2 and Line 4 and re-allocates a new object to the dereferenced memory region, which will evade our protection. ViK$_O$ will still mitigate the exploit if the pointer is dereferenced again in other functions later, as we have observed in CVE-2019-2215.

Mitigating real-world kernel exploits

To evaluate the effectiveness of ViK, we selected five known UAF vulnerabilities with public exploits on Linux kernel and tested them against a ViK-protected Linux kernel 4.12. All these vulnerabilities are related to race conditions. For Android kernel, we picked four UAF vulnerabilities, three out of which are caused by race conditions. All the exploits are collected from the Exploit Database and another research project FUZE [6, 143]. Five of these vulnerabilities (CVE-2017-17053, -15649, CVE-2019-2215, -2025, and -2000) can be exploited directly on Linux kernel 4.12 and Android

		Linux kernel 4.12		
CVE	**Race Condition**	**ViK$_S$**	**ViK$_O$**	**(ViK$_{TBI}$)**
CVE-2017-17053	Yes	✓	✓	(✓)
CVE-2017-15649	Yes	✓	✓	(✓)
CVE-2017-11176	Yes	✓	✓	(✓*)
CVE-2017-2636	Yes	✓	✓	(✓)
CVE-2016-8655	Yes	✓	✓	(✓)
CVE-2016-4557	Yes	✓	✓	(✓)
		Android kernel 4.14		
CVE	**Race Condition**	**ViK$_S$**	**ViK$_O$**	**ViK$_{TBI}$**
CVE-2019-2215	No	✓	✓	✗
CVE-2019-2025	Yes	✓	✓	✓
CVE-2019-2000	Yes	✓	✓	✓*
CVE-2017-7533	Yes	✓	✓	✓

***:** ViK$_{TBI}$ did not immediately stop the exploit when UAF happened, but it stopped the attack through a delayed mitigation.

Table 6.3: Experimental Results of ViK Defending Agianst Known UAF Exploits in OS Kernels. The ViK$_{TBI}$ results for Linux kernel are manually synthesized.

kernel 4.14 (without ViK), while the other four vulnerabilities (CVE-2017-11176, -7533, -2636, CVE-2016-8655, and -4817) do not exist on our versions of Linux and Android kernels. We manually ported them onto Linux kernel 4.12 and Android Kernel 4.14 by reverting the related patches. Details of the selected vulnerabilities and the results of the security evaluation are shown in Table 6.3. As expected, ViK-protected kernels, including ViK$_O$, detected UAFs caused by these vulnerabilities.

TBI Optimization

We evaluated the TBI-featured variant of ViK, ViK$_{TBI}$, on Android kernel 4.14 and list the results in Table 6.3. ViK$_{TBI}$ did not stop the exploit for CVE-2019-2215 because the exploit uses a pointer that points to the middle of an object while ViK$_{TBI}$ only inspects pointers that point to the base address of an object. Also, it is worth noting that a delayed mitigation happened with CVE-2019-2000: Since the dangling pointer used in the exploit points to the middle of an object, ViK$_{TBI}$ did not detect the UAF exploit when this pointer is first dereferenced and the victim object is updated. However, ViK$_{TBI}$ detected the UAF when the original pointer (which points to the base address of the object) is later used before returning from the kernel to user space, which illustrates the effectiveness of ViK even when applying all aforementioned optimizations. Finally, since current x86-64 CPUs do not implement TBI, we examined every Linux kernel vulnerability in our dataset, manually analyzed if ViK$_{TBI}$ will defend against each UAF exploit, and present the results in the ViK$_{TBI}$ column of Linux kernel in Table 6.3.

Mitigating user-space UAF exploits

We also conducted experiments for user-space ViK on C programs to demonstrate the effectiveness of ViK. As shown in Table 6.4, ViK prevented all tested UAF vulnerabilities in user-space programs except for the case of libpng (CVE-2019-7317) under ViK$_O$. We manually examined the vulnerability and found that the UAF is caused by an early call of `free()` before the second dereference of a UAF-unsafe pointer in the same function and the same thread, which meets an optimization condition that is used in ViK$_O$. Since the program crashes immediately after the second pointer dereference (which makes this bug not exploitable), delayed mitigation in ViK$_O$ did

CVE	Applications	ViK$_S$	ViK$_O$
CVE-2019-7317	libpng	✓	✗*
CVE-2018-9009	libming	✓	✓
CVE-2016-7835	H2O server	✓	✓
CVE-2016-4817	H2O server	✓	✓

***:** CVE-2019-7317 is not exploitable; The target crashed immediately before delayed mitigation had a chance to happen in ViK$_O$.

Table 6.4: Experimental results of ViK defending against known UAF vulnerabilities in user-space programs.

not happen. While this failure does not impact the security, we still choose to report it as a failure case of ViK for integrity.

6.9.4 Performance: ViK-protected Kernels

For performance evaluation, we used micro benchmarks as implemented in two renowned benchmark tools—LMbench and UnixBench—on Ubuntu and Android. We then evaluated the memory overhead of each kernel when protected by ViK. Finally, we measured the performance impact that ViK-protected kernels have on user-space programs.

Micro benchmarks on kernels

We first present the benchmark results using LMbench which measures latency and basic costs of key operations of UNIX/POSIX systems [101]. Table 6.5 shows the results for each kernel. In ViK$_O$, the average percentages of increased latency are 20.71% and 19.86% on the Ubuntu and Android kernel, respectively. Because ViK

	Linux kernel 4.12		Android kernel 4.14	
Benchmark	**ViK$_S$**	**ViK$_O$**	**ViK$_S$**	**ViK$_O$**
Latency (percentage of increase)				
Simple syscall	16.88%	10.82%	15.60%	7.16%
Simple fstat	96.74%	67.41%	68.86%	47.15%
Simple open/close	140.40%	77.01%	74.88%	38.62%
Select on fd's	23.19%	15.42%	35.52%	28.47%
Sig. handler installation	6.36%	4.09%	19.24%	6.37%
Sig. handler overhead	41.19%	4.34%	113.83%	46.86%
Protection fault	0%	0%	5.52%	0%
Pipe	40.91%	26.48%	60.80%	15.45%
AF_UNIX sock stream	26.91%	8.35%	77.91%	23.80%
Process fork+exit	85.90%	68.01%	35.13%	16.40%
Process fork+/bin/sh -c	96.45%	62.66%	32.21%	14.31%
GeoMean	**40.77%**	**20.71%**	**37.13%**	**19.86%**

Table 6.5: The Runtime Overhead Measured by LMbench on ViK-protected OS Kernels.

inserts fewer *inspect*() functions into the Android kernel, its performance overhead is lower than ViK on the Linux kernel 4.12. As expected, ViK$_O$ has substantially better performance. Compared with ViK$_S$, on both kernels, ViK$_O$ exhibits about 20% lower performance overhead.

We also evaluated ViK-protected kernels using UnixBench [14]. UnixBench includes multiple benchmarks that test the performance of a UNIX-like system. It generates a system index score as an overall indicator of the performance. As shown in Table 6.6, the results are similar to the average percentages of increased latency in Table 6.5. In summary, micro benchmark results show that the ViK-protected OS kernels incur around 22% and 20% runtime overhead on Android and Linux kernels,

	Linux kernel 4.12		Android kernel 4.14	
Benchmark	**ViK$_S$**	**ViK$_O$**	**ViK$_S$**	**ViK$_O$**
Dhrystone 2	0%	0%	0%	0%
DP Whetstone	0.83%	0.21%	0%	0%
Execl Throughput	77.95%	48.18%	50.32%	28.62%
File Copy 1024 bufsize	100.30%	56.43%	123.00%	61.13%
File Copy 256 bufsize	99.33%	54.45%	148.91%	77.51%
File Copy 4096 bufsize	70.71%	41.89%	71.42%	34.01%
Pipe Throughput	110.90%	74.66%	60.77%	41.55%
Pipe-based Ctxt. Switching	126.70%	80.78%	50.09%	0.39%
Process Creation	85.05%	57.22%	42.53%	22.58%
Shell Scripts (1 concurrent)	58.47%	36.16%	34.88%	22.13%
Shell Scripts (8 concurrent)	55.96%	35.71%	27.24%	16.02%
System call overhead	8.89%	1.11%	30.18%	15.45%
GeoMean	**45.14%**	**22.20%**	**54.80%**	**19.80%**

Table 6.6: The Performance Overhead Measured by UnixBench on ViK-protected OS Kernels.

respectively.

Additionally, we compare ViK against other kernel defenses, specifically KENALI [130] and KCoFI [56], using the results of LMbench. KENALI, which enforces data-flow integrity (DFI) over distinguished regions, and KCoFI, which enforces control-flow integrity (CFI) on the entire kernel, incur average overhead of about 105% and 137%, respectively. In comparison, ViK incurs average overhead of around 21% on both the Linux kernel and the Android kernel. However, conducting a fair comparison with KENALI and KCoFI is difficult, because they use different kernels and perform evaluations on different processors. Hence, this comparison is only provided as a rough estimate of ViK's performance on OS kernels.

	After Reboot (%)		After Bench (%)	
Memory alignment	**Ubuntu**	**Android**	**Ubuntu**	**Android**
Table 6.1	13.08%	16.01%	25.03%	28.30%
64 bytes	42.42%	43.98%	41.69%	43.89%

Table 6.7: Memory Overhead Imposed by ViK on Each Kernel.

Memory overhead

To measure the memory overhead of ViK-protected kernels, we checked the total amount of memory used by each kernel in `/proc/meminfo`. We measured the memory usage (1) after the system finished booting, and (2) after running LMbench and report the results in Table 6.7. When ViK aligned memory addresses by 64 bytes, the overall memory overhead was around 42%. ViK achieved much lower memory overhead when it employed the alignment strategy as described in Table 6.1 where 16-byte alignment is used for objects smaller than 256 bytes and 64-byte alignment is used for other objects. There is no difference in memory usage between ViK_S and ViK_O mode because they allocate the same number of objects. The major source of memory overhead is the amount of memory added to structs to guarantee the alignment. In our implementation of ViK, we used the constants shown in Table 6.1. For lower memory overhead, ViK will need various sets of M and N that are optimally calculated for different sizes of kernel objects, which requires a more complex implementation.

Performance of ViK_{TBI}

Table 6.8 shows the runtime and memory overhead of ViK_{TBI}. The use of TBI and the reduced number of inspection functions make the runtime overhead of ViK_{TBI} negligible (<3%). The memory overhead of ViK_{TBI} is low: 8% after booting and 17%

Android kernel 4.14 — ViK$_{TBI}$			
UnixBench benchmarks	**Overhead**	**LMbench benchmarks**	**Overhead**
Dhrystone 2	0%	Simple syscall	0%
DP Whetstone	0%	Simple fstat	0%
Execl Throughput	0%	Simple open/close	0.9%
File Copy 1024 bufsize	1.0%	Select on fd's	0.2%
File Copy 256 bufsize	6.3%	Sig. handler installation	0%
File Copy 4096 bufsize	0%	Sig. handler overhead	0%
Pipe Throughput	0%	Protection fault	0%
Pipe-based Ctxt. Switching	0%	Pipe	0%
Process Creation	1.1%	AF_UNIX sock stream	2.1%
Shell Scripts (1 concurrent)	0%	Process fork+exit	0%
Shell Scripts (8 concurrent)	0%	Process fork+/bin/sh -c	0%
System Call Overhead	0%		
GeoMean	**1.91%**	**GeoMean**	**0.72%**
Memory overhead			
After Reboot	7.80%	After Bench	17.50%

Table 6.8: The Performance and Memory Overhead on ViK$_{TBI}$–protected Android Kernel.

after running benchmarks. The performance of ViK$_{TBI}$ has convinced a major hardware vendor that it has sufficiently low overhead to be deployed on customer-facing devices. This vendor will begin delivering devices with ViK$_{TBI}$-protected Android kernels in late 2020.

6.9.5 Performance: ViK-protected User-space Programs

We evaluated the user-space ViK implementation by measuring the performance of ViK-protected C and C++ programs. To evaluate the runtime and memory overhead

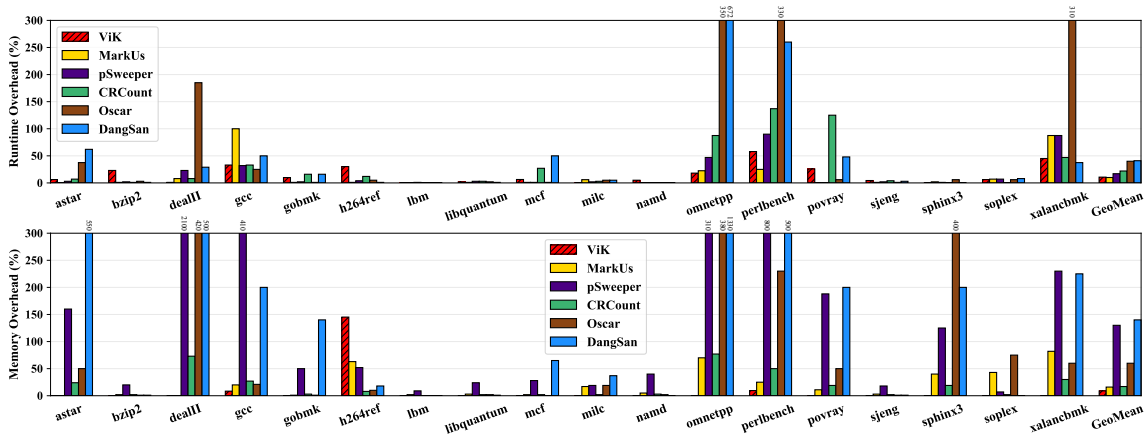


Figure 6.5: Runtime and Memory Overhead Comparison of the User-space Implementation of ViK with Previous Approaches (MarkUs, pSweeper, CRCount, Oscar, and DangSan) on the SPEC CPU 2006 Benchmark Programs.

of ViK$_O$ (16-byte aligned), We converted the LLVM instrumentation pass into a Link-Time Optimization (LTO) module and compiled the programs with the LTO module enabled. The experimental results are shown in Figure 6.5. For clarity, we also include in the figure the overhead numbers of applying five state-of-the-art user-space runtime UAF protections (MarkUs, pSweeper, CRCount, Oscar, and DangSan) on the same programs.[4]

Runtime overhead

ViK has average runtime overhead of 10.6% (MarkUs has the same overall overhead within rounding error). However, ViK performs better than the other five defenses when we compare the average overhead on the most pointer-intensive 8 benchmarks in terms of the number of memory allocations and pointer operations (`perlbench`, `omnetpp`, `mcf`, `gcc`, `povray`, `milc`, `xalancbmk`, `astar`, `soplex`, and `gobmk`)—ViK in-

[4]The performance numbers of MarkUs, pSweeper, CRCount, Oscar, and DangSan are extracted from the original papers or provided by their authors.

curs average runtime overhead of about 20%, while it is 25% for MarkUs, 27% for pSweeper, 48% for CRCount, 107% for Oscar, and 128% for DangSan. Compared to other defenses, ViK shows better or similar runtime overhead on all but two programs, which are `bzip2` and `h264ref`. This is because in ViK, pointer dereferences have a larger impact on the runtime overhead than memory allocations or deallocations. These two programs have relatively low numbers of memory allocations and deallocations but possess high numbers of pointer dereferences, which are unideal for ViK. For example, during an execution of `bzip2`, the `malloc` function executed 8 times at the beginning of its compression routine and 6 times at the beginning of its decompression routine, which are much lower than the numbers for other programs.

Memory overhead

We measured the memory overhead of ViK-protected user-space programs by taking the maximum resident set sizes (RSS). ViK incurs average memory overhead of about 9%, compared with 16% for MarkUs, 130% for pSweeper, 17% for CRCount, 60% for Oscar, 140% for DangSan. Overall, ViK achieves lower memory overhead than the other solutions on all tested programs except for `h264ref`. We found that the majority of memory allocations in `h264ref` are small-sized, which severely penalizes ViK due to its memory alignment enforcement. Our theory is supported by ViK's performance on the most four allocation-intensive benchmarks `perlbench`, `xalancbmk`, `omnetpp`, and `dealII`—ViK incurs much less memory overhead (2.42%) than the others (about 40% for MarkUs and 50% for CRCount).

Static analysis.

Through static analysis, ViK finds pointer operations using UAF-unsafe pointers in a flow- and path-sensitive manner. We bypass common challenges of static analysis (*e.g.*, scalability) by limiting the range of static analysis to individual modules. However, this design decision limits the potential of ViK's optimizations. We expect ViK to have even lower runtime overhead without sacrificing the security guarantees if we can apply inter-procedural and inter-modular optimizations.

Implementation details.

ViK works on LLVM IR, which provides both advantages (e.g., multi-architecture support) and drawbacks, one of which is that ViK cannot instrument inline assembly and requires manual modification. Also, ViK requires a preliminary analysis to determine the optimal constants (M and N), and it does not support using multiple sets of constants for different base identifiers, which we deem as engineering effort.

Arbitrary memory read and write.

Arbitrary memory read and write vulnerabilities in user-space programs and kernels may allow attackers to tamper the internal state of ViK, which enables Object ID forging. We argue that such vulnerabilities are rare in real world. These vulnerabilities can be addressed by other defenses. Currently, ViK does not defend against such vulnerabilities.

Access validation

Some prior mitigations validate memory accesses in ways that are similar to ViK [129, 106]. Although they have shown to be robust against UAF exploits, they all introduce significant runtime overhead and false positives. Their high runtime overhead stems from the workload that these defenses introduce for managing and checking metadata, which happens on *every pointer propagation and dereference*. In contrast, ViK does not track pointer propagation and has much lower runtime overhead. Moreover, these defenses have compatibility issues that ViK does not have: CETS does not support multithreading [106], and MemSafe requires performing a full-program data-flow analysis [129] which is likely infeasible for OS kernels and large, complex programs.

Pointer invalidation

Many systems attempt to detect the creation of dangling pointers by tracking reference relationships between pointers and objects [126, 134, 89, 153, 48]. Their designs differ regarding the format of the metadata and the manner in which the metadata is managed. CRCount uses a pointer bitmap to represent locations of heap pointers for reference counting [126]. pSweeper invalidates dangling pointers through another thread that runs in the background, managing a live pointer list and sweeping dangling pointers [92]. DangSan employs an append-only per-thread pointer logger for each memory object [134]. DANGNULL records the relationship between objects in a hierarchical structure called shadowObjTree [89]. However, common to all approaches is the existence of the joint metadata, which imposes high runtime and memory overhead, especially for multithreaded programs. Additionally, these approaches

suffer from propagations of type-unsafe pointers and non-pointer type variables that have pointer values, because of the difficulty of achieving complete data flow analysis [126, 134, 89, 153]. This can lead to mistakes in correctly maintaining reference relationships.

Safe memory allocation

Another type of UAF mitigation is to prevent reusing unsafe address spaces when a new object is allocated [57, 128, 45, 108, 19]. These approaches suffer from high memory overhead caused by their object allocation or memory management policies. Recently, Markus [18] proposed an approach to verify deallocations through a marking procedure for finding live objects, which showed good memory and performance overhead in user-space programs. However, ViK has much lower memory overhead in allocation-intensive programs (2.42%) than Markus (about 40%) and the approach is not intended to be used by OS kernels.

Static UAF detection

Several static analysis approaches are proposed for finding UAF errors, such as Tac, CRed, and DCUAF [44, 149, 148]. Although they showed some effectiveness in finding possible errors for user-space applications [149, 148] and device drivers [44], they suffer from both high false positives and false negatives due to unsound handling of complex conditions and paths, and inaccurate pointer analysis.

Hardware-based approach

WatchdogLite proposes a new instruction set (Intel's Instruction Set Architecture extension) for preventing out-of-bound accesses and UAF errors through compiler support [105]. BOGO utilizes bounds metadata managed by the Intel MPX for pro-

viding temporal memory safety [156]: When memory is deallocated, BOGO checks the bound metadata and invalidates dangling pointers. Although both approaches heavily rely on hardware features, they all impose significant runtime overhead (average slowdown of 29% for Watchdog and 60% for BOGO on SPEC CPU 2006 benchmarks).

Memory error detection

The high runtime overhead of runtime mitigations has led to the emergence of general memory error detectors. AddressSanitizer (ASan) is a popular tool for user-space applications [123]. However, it cannot detect all UAF errors, especially when memory objects are reallocated onto previously freed memory regions [89]. The Kernel Address Sanitizer (KASan) is a memory error detector designed for the kernel [8]. KASan provides provides two modes: generic KASan and software-tag-based KASan. The software-tag-based KASan employs the unused bits (similar to ViK) with the top byte ignore (TBI) feature of AArch64 CPUs, and thus, it only supports the AArch64 architecture. KASan uses shadow memory to check whether each memory access is safe or not, which imposes huge memory overhead. Also, KASan inserts a checker for each memory access, which introduces high runtime overhead of about 200%. Hardware-assisted AddressSanitizer (HWASan) also uses the TBI feature of ARM [124]. Similar to the software-tag-based KASan, HWASan has prohibitive runtime overhead (roughly 200%), but with lower memory overhead (up to 35%). Currently, HWASan can only be used for user-space applications on a modified kernel, which limits its applicability. In comparison, ViK is a runtime mitigation that functions in both OS kernels and user-space programs without architectural dependencies.

Chapter 7

CONCLUSION

The diversity and complexity of computing devices and software are rapidly increasing as our lives are increasingly dependent on computers. Therefore, we must ensure that the software running on computing devices secure. Unfortunately, we are observing that this is not the case, rather, vulnerabilities are abundant. Especially, nowadays, these expanded targets have brought vulnerabilities which are increasing every year. Also, attackers have found new classes of exploits using microarchitectural features such as CPU cache side-channels. This ubiquitous computing environment is becoming more vulnerable, and, cybersecurity is becoming more critical than ever. Therefore, exploiting and mitigating emerging security vulnerabilities are of great importance for securing software systems used in our daily lives.

Throughout the four chapters presented in this dissertation, I have discussed how we can exploit and mitigate two different types of advanced security vulnerabilities to assess the evolving threat landscape and enhance state-of-the-art defenses.

Building Data Covert Channels

We presented cross-world covert channel attacks on ARM TrustZone, which is designed to provide hardware-assisted isolation. We demonstrated that existing channel protection solutions, such as SeCReT, or even more powerful mechanisms, such as a strong monitor, can be bypassed. We discussed the reasons why previous attacks, including Prime+Probe and Flush+Reload, do not work for the cross-world scenario on ARM. Lastly we designed a low noise, no shared memory needed cache attack named Prime+Count by leveraging overlooked PMU "L1/L2 cache refill

events." We also presented how to utilize PRIME+COUNT to build fast cross-world covert channels in ARM TrustZone architecture.

Mitigating Cache Side-channel Attacks

In this work, we presented SMOKEBOMB: a novel, systematic software approach to defeat cache side-channel attacks on the ARM architecture. Our approach can protect access patterns on sensitive data from attackers easily by automatically providing the protection mechanism to applications as a compiler extension. The experimental results show that SMOKEBOMB protects sensitive information leakages against cache attack methods known to us effectively.

Leaking Sensitive Information of OS Kernels

We proposed a generic approach to exploit uses of uninitialized stack data in Linux kernels to leak pointer values that are pointing to either kernel functions or to the kernel stack. These leaked pointer values can then be used to defeat KASLR and mount future attacks against Linux kernels. Our evaluation results show that we can effectively analyze and exploit stack-based information-leak vulnerabilities through the proposed approach. Our proposed approach exposes the actual exploitability and severity of information disclosure bugs in Linux kernels and will raise awareness of the community on the security impact of these bugs. We expect our findings will help adjust CVSS scoring for information leak bugs inside Linux kernels.

Assuring the Temporal Memory Safety

Temporal memory safety violations are critical and it is challenging to enforce a temporal memory safety in an efficient, strong, and flexible (widely applicable) manner. Even though access validation approaches can provide a strong defense, their common

problem is practicability due to high-performance overhead and many false positives. In this work, we proposed a novel defense, ViK, that detects UAF exploits with no false positives. Also, as our evaluation indicates, ViK imposes low overhead, and is a practical defense.

[1] *Automatic variable initialization*. https://reviews.llvm.org/D54604.

[2] *CVE (Vulnerability) Details: CVE-2016-4486*. https://www.cvedetails.com/cve/CVE-2016-4486.

[3] *CVE (Vulnerability) Details: CVE-2016-4569*. https://www.cvedetails.com/cve/CVE-2016-4569.

[4] *CVE (Vulnerability) Details: CVE-2018-11508*. https://www.cvedetails.com/cve/CVE-2018-11508.

[5] *Arm A-Profile Architecture Developments: Armv8.5-A*, 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[6] *Exploit Database*, 2019. https://www.exploit-db.com.

[7] *KASAN: remove use after scope bugs detection.*, 2019. https://kernel.googlesource.com/pub/scm/linux/kernel/git/torvalds/linux/+/7771bdbbfd3d6f204631b6fd9e1bbc30cd15918e.

[8] *The Kernel Address Sanitizer (KASAN)*, 2019. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[9] *White Paper: ARM v8.5-A Memory Tagging Extension*, 2019. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf?revision=ef3521b9-322c-4536-a800-5ee35a0e7665&la=en&hash=D9F2FA87FEA090C2B20938F09BBAC71698FA18BA.

[10] *KernelMemorySanitizer, a detector of uses of uninitialized memory in the Linux kernel*, (accessed Feb 2nd, 2020). https://github.com/google/kmsan.

[11] *Linux Kernel: Vulnerability Statistics*, (accessed Feb 2nd, 2020). https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[12] *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*, (accessed Feb 2nd, 2020). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[13] *Attribute Syntax*, (accessed Jan 29, 2019). https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html.

[14] *Byte-unixbench: A Unix benchmark suite*, (accessed Jan 29, 2019). https://github.com/kdlucas/byte-unixbench.

[15] *Moz's list of the top 500 domains and pages on the web*, (accessed Jan 29, 2019). https://moz.com/top500.

[16] Josh Aas. *Understanding the Linux 2.6. 8.1 CPU scheduler*, 2005.

[17] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA)*, pages 256–273, San Fancisco, CA, April 2008.

[18] Sam Ainsworth and Timothy M Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[19] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, pages 177–192, Washingtion, DC, August 2010.

[20] ARM. *Address spaces in Armv8-A*. `https://developer.arm.com/architectures/learn-the-architecture/memory-management/address-spaces-in-armv8-a`.

[21] ARM. *Cortex-A8 Technical Reference Manual*, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html`.

[22] ARM. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html`, 2012.

[23] ARM. ARMv6-M Architecture Reference Manual. `https://silver.arm.com/download/download.tm?pv=1102513`, 2012.

[24] ARM. *Cortex-A9 Technical Reference Manual*, 2012. `http://infocenter.arm.com/help/topic/com.arm.doc.100511_0401_10_en/index.html`.

[25] ARM. *Cortex-A15 MPCore Processor Technical Reference Manual*, 2013. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/index.html`.

[26] ARM. *Cortex-A7 MPCore Processor Technical Reference Manual*, 2013. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/index.html`.

[27] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2014. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html`.

[28] ARM. *Cortex-A17 MPCore Processor Technical Reference Manual*, 2014. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0535c/index.html`.

[29] ARM. *ARM Cortex-A Series Programmerś Guide for ARMv8-A*, 2015. `http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/index.html`.

[30] ARM. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.k/index.html`, 2016.

[31] ARM. *ARMv8-A Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487b.b/index.html.

[32] ARM. *Cortex-A5 MPCore Processor Technical Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0434c/index.html.

[33] ARM. *Cortex-A53 MPCore Processor Technical Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/index.html.

[34] ARM. *Cortex-A57 MPCore Processor Technical Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/index.html.

[35] ARM. *Cortex-A72 MPCore Processor Technical Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.100095_0003_06_en/index.html.

[36] ARM. *Cortex-A73 MPCore Processor Technical Reference Manual*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.100048_0002_05_en/index.html.

[37] ARM. SMC CALLING CONVENTION System Software on ARM Platforms. http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf, 2016.

[38] ARM. *AMBA Protocol*, 2017. https://developer.arm.com/products/architecture/amba-protocol.

[39] ARM. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware, 2017.

[40] ARM. *Cortex-A32 Processor Technical Reference Manual*, 2017. http://infocenter.arm.com/help/topic/com.arm.doc.100241_0001_00_en/index.html.

[41] ARM. *Cortex-A35 Processor Technical Reference Manual*, 2017. http://infocenter.arm.com/help/topic/com.arm.doc.100236_0002_00_en/index.html.

[42] ARM. *Cortex-A55 MPCore Processor Technical Reference Manual*, 2017. http://infocenter.arm.com/help/topic/com.arm.doc.100442_0100_00_en/index.html.

[43] ARM. *Cortex-A75 Core Technical Reference Manual*, 2017. http://infocenter.arm.com/help/topic/com.arm.doc.100403_0200_00_en/index.html.

[44] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 255–368, Renton, WA, July 2019.

[45] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, Ottawa, Canada, June 2006.

[46] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, 2017.

[47] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *Proceedings of the 15th the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 667–684, Tokyo, Japan, December 2009.

[48] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–143, Minneapolis, MN, July 2012.

[49] Serdar Cabuk, Carla E Brodley, and Clay Shields. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 178–187, Washington, DC, October 2004.

[50] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, Shanghai, China, July 2011.

[51] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade attack on trustzone. *arXiv preprint arXiv:1707.05082*, 2017.

[52] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 1707–1722, London, UK, October 2019.

[53] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[54] Kees Cook. *Kernel exploitation via uninitialized stack*, 2011. `https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf`.

[55] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[56] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, pages 292–307, San Jose, CA, May 2014.

[57] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 815–832, Vancouver, BC, Canada, August 2017.

[58] CVE Details. *Linux: Vulnerability Statistics*, 2019. `https://www.cvedetails.com/vendor/33/Linux.html`.

[59] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 51–67, Vancouver, BC, Canada, August 2017.

[60] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[61] Halvar Flake. *Attacks on uninitialized local variables*, 2006. `https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf`.

[62] Vincenzo Frascino. ARM v8.5 Memory Tagging Extension. In *Linux Plumbers Conference 2019*, Lisbon, Portugal, September 2019.

[63] Shekhar R Gaddam, Vir V Phoha, and Kiran S Balagani. K-Means+ ID3: A novel method for supervised anomaly detection by cascading K-Means clustering and ID3 decision tree learning methods. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):345–354, 2007.

[64] Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz. Static Detection of Uninitialized Stack Variables in Binary Code. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, pages 68–87, Luxembourg, September 2019.

[65] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[66] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 1075–1091, Vancouver, BC, Canada, August 2017.

[67] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 300–321, San Sebastian, Spain, July 2016.

[68] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 279–299, San Sebastian, Spain, July 2016.

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

[70] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Security Symposium (Security)*, pages 897–912, Washington, DC, August 2015.

[71] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, pages 38–55, San Jose, CA, May 2016.

[72] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, pages 490–505, Oakland, CA, May 2011.

[73] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 299–312, Santa Clara, CA, July 2017.

[74] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898, 2015. `https://eprint.iacr.org/2015/898`.

[75] Intel. *5-Level Paging and 5-Level EPT*. `https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf`.

[76] RISC-V International. *RISC-V Instruction Set Manual*, 2019. `https://github.com/riscv/riscv-isa-manual`.

[77] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ A: A shared cache attack that works across cores and defies VM sandboxing-and its application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, pages 591–604, San Jose, CA, May 2015.

[78] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 353–364, Xi'an, China, May–June 2016.

[79] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 299–319, Gothenburg, Sweden, September 2014.

[80] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[81] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 380–392, Vienna, Austria, October 2016.

[82] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[83] The kernel development community. *The Linux Kernel 5.9.0-rc3 documentation: Application Data Integrity (ADI)*, 2019. `https://www.kernel.org/doc/html/latest/sparc/adi.html`.

[84] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (Security)*, pages 189–204, Bellevue, WA, August 2012.

[85] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the the 2nd ACM workshop on Computer security architectures*, pages 25–34, Alexandria, VA, October 2008.

[86] laginimaineb. Exploit that extracts Qualcomm's KeyMaster keys using CVE-2015-6639. https://github.com/laginimaineb/ExtractKeyMaster, 2016.

[87] laginimaineb. Qualcomm TrustZone kernel privilege escalation using CVE-2016-2431. https://github.com/laginimaineb/cve-2016-2431, 2016.

[88] Paul Larson. Testing linux with the linux test project. In *Proceedings of the Linux Symposium 2002*, Ottawa, Canada, June 2002.

[89] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[90] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, J Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 781–797, Santa Clara, CA, August 2019.

[91] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium (Security)*, pages 549–564, Austin, TX, August 2016.

[92] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 1635–1648, Toronto, Canada, October 2018.

[93] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, Barcelona, Spain, March 2016.

[94] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[95] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, pages 605–622, San Jose, CA, May 2015.

[96] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 920–932, Vienna, Austria, October 2016.

[97] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[98] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[99] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[100] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993*, pages 259–270, San Diego, CA, January 1993.

[101] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, pages 279–294, San Diego, CA, January 1996.

[102] Matt Miller. Trends, challenge, and strategic shifts in software vulnerability mitigation. In *BlueHat IL 2019*, Tel Aviv, Israel, February 2019.

[103] MIPS. *MIPS Architecture For Programmers Volume III: MIPS64/microMIPS64TM Privileged Resource Architecture*, 2015. `https://www.mips.com/?do-download=the-mips64-and-micromips64-privileged-resource-architecture-v6-03`.

[104] MITRE. CVE-2013-3051 Detail. https://nvd.nist.gov/vuln/detail/CVE-2013-3051, 2013.

[105] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, Orlando, FL, February 2014.

[106] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices*, 45(8):31–40, 2010.

[107] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. Position paper: Challenges towards securing hardware-assisted execution environments. In *Proceedings of the the Hardware and Architectural Support for Security and Privacy*, Toronto, Canada, June 2017.

[108] Gene Novark and Emery D Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 573–584, Chicago, IL, October 2010.

[109] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Security Symposium (Security)*, pages 619–636, Austin, TX, August 2016.

[110] OP-TEE. OP-TEE Trusted OS Documentation. `https://www.op-tee.org/`, 2017.

[111] Oracle. *Oracle Solaris 11.3 Programming Interfaces Guide: Using Application Data Integrity (ADI)*, 2019. https://docs.oracle.com/cd/E53394_01/html/E54815/gqajs.html.

[112] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 559–572, Providence, RI, April 2019.

[113] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA)*, pages 1–20, San Jose, CA, February 2006.

[114] Salva Peiró, M Muñoz, Miguel Masmano, and Alfons Crespo. Detecting stack based kernel information leaks. In *Proceedings of the International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, pages 321–331, Bilbao, Spain, June 2014.

[115] Alexander Popov. STACKLEAK: A Long Way to the Linux Kernel Mainline. In *Linux Security Summit 2018*, Vancouber, Canada, August 2018.

[116] Alexander Potapenko. Dealing with Uninitialized Memory in the Kernel. In *Linux Security Summit Europe 2019*, Lyon, France, October–November 2019.

[117] Qualcomm. *Pointer Authentication on ARMv8.3*, 2017. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[118] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[119] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[120] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, Chicago, IL, November 2009.

[121] Dan Rosenberg. Unlock the Motorola Bootloader. http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html, 2013.

[122] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 279–299, Bonn, Germany, July 2017.

[123] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, Boston, MA, June 2012.

[124] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety. *ArXiv e-prints*, February 2018.

[125] Gaurav Shah, Andres Molina, Matt Blaze, et al. Keyboards and Covert Channels. In *Proceedings of the 15th USENIX Security Symposium (Security)*, pages 59–75, Vancouver, Canada, July 2006.

[126] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C+. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[127] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, pages 138–157, San Jose, CA, May 2016.

[128] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 2389–2403, Dallas, TX, October–November 2017.

[129] Matthew S Simpson and Rajeev K Barua. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[130] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[131] Inc. Sun Microsystems. *OpenSPARC T2 Core Microarchitecture Specification*, 2007. `https://www.oracle.com/technetwork/systems/opensparc/t2-06-opensparct2-core-microarch-1537749.html`.

[132] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, pages 48–62, San Francisco, CA, May 2013.

[133] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[134] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 405–419, Belgrade, Serbia, April 2017.

[135] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proceedings of the 24th USENIX Security Symposium (Security)*, pages 913–928, Washington, DC, August 2015.

[136] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proceedings of the 24th USENIX Security Symposium (Security)*, pages 913–928, Washington, DC, August 2015.

[137] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 235–252, Vancouver, BC, Canada, August 2017.

[138] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, July 2012.

[139] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 473–482, Miami, FL, December 2006.

[140] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, Como, Italy, November 2008.

[141] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual*, 2017. `https://github.com/riscv/riscv-isa-manual`.

[142] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 1187–1204, Santa Clara, CA, August 2019.

[143] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, pages 781–797, Baltimore, MD, August 2018.

[144] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (Security)*, pages 159–173, Bellevue, WA, August 2012.

[145] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 414–425, Denver, CO, November 2015.

[146] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, and Richard Hiltunen, Mattia nd Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40, Chicago, IL, October 2011.

[147] Zhenquan Xu, Gongshen Liu, Tielei Wang, and Hao Xu. Exploitations of uninitialized uses on macos sierra. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, Canada, August 2017.

[148] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 42–54, Orlando, FL, December 2017.

[149] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337, Gothenburg, Sweden, May–June 2018.

[150] Yongcheol Yang, Jiyoung Moon, Kiuhae Jung, and Jeik Kim. Downloadable trusted applications on Tizen TV: TrustWare Extension: As a downloadable application framework. In *Proceedings of the 2018 IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, January 2018.

[151] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[152] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, pages 719–732, San Diego, CA, August 2014.

[153] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[154] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. `https://eprint.iacr.org/2016/980.pdf`, 2016.

[155] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[156] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, Providence, RI, April 2019.

[157] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 858–870, Vienna, Austria, October 2016.

[158] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 858–870, Vienna, Austria, October 2016.

[159] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, pages 313–328, Oakland, CA, May 2011.

[160] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, pages 313–328, Oakland, CA, May 2011.

[161] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, Raleigh, NC, October 2012.

[162] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, Raleigh, NC, October 2012.

[163] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 990–1003, Scottsdale, Arizona, November 2014.

[164] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 990–1003, Scottsdale, AZ, November 2014.

[165] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 827–838, Berlin, Germany, October 2013.

[166] YongBin Zhou and DengGuo Feng. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.

[167] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 871–882, Vienna, Austria, October 2016.