UNIVERSITY OF CALIFORNIA
RIVERSIDE

CPU Side-Channels: New Attacks and Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Daimeng Wang

March 2020

Dissertation Committee:

Dr. Zhiyun Qian, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Srikanth V. Krishnamurthy
Dr. Chengyu Song

The Dissertation of Daimeng Wang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to express my deepest appreciation to my committee. I'm extremely grateful to the committee chair Dr. Zhiyun Qian. As my teacher and mentor, he without doubt, set a shining role model and provided me with unparalleled patience and extensive guidance throughout the duration of my PhD, especially in my early years of struggling. Dr. Nael Abu-Ghazaleh and Dr. Srikanth V. Krishnamurthy have been co-authors on all of my major publications. The completion of my dissertation would not have been possible without their invaluable expertise, ingenious insights, and heart-warming encouragements. I also had great pleasure of working with Dr. Chengyu Song briefly as a teaching assistant to the undergrad OS class, which has been a wonderful and fruitful experience.

Further, I would like to extend my thanks to Dr. Edward J.M. Colbert and Dr. Paul Yu, for their profound belief in my work. I'd like to recognize the assistance of my colleagues, most notably Dr. Ajaya Neupane, who played an instrumental role in the keystroke attack project, as well as Zheng Zhang and Hang Zhang, who extended a great amount of assistance in the kernel fuzzing project. There is also everyone else in the lab, between whom the exchange of ideas produces numerous sparks of inspiration.

Particularly helpful to me during my program are my parents, who always supported and nurtured me during my PhD program. Their occasional visit to the US provided me with relief that cannot be overestimated. I cannot forget to also thank friends for all their unconditional support and the color they bring to my life. Finally, thanks should also go to my two feline companions Flash and Snow. They have accompanied me through countless lonely nights and provided me with great comfort and serenity.

The text of this dissertation, in part or in full, is a reprint of the material as it appears in *PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack* [99] and *Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries* [98]. The co-authors listed in these publications provided technical expertise to the research which forms the basis for this dissertation.

ABSTRACT OF THE DISSERTATION

CPU Side-Channels: New Attacks and Applications

by

Daimeng Wang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2020
Dr. Zhiyun Qian, Chairperson

CPU micro-architectural side-channels, or CPU side-channels in short, have gained plenty of attention recently. Many existing works have proved that classical CPU side-channel attacks (e.g. *Prime+probe* [81] and *Flush+reload* [105]) as well as recently-discovered attacks (e.g. *Spectre* [65], *Meltdown* [71], *Zombieload* [90]) are practical and effective against cryptographic libraries. However, it's in our belief that CPU side-channels have more potential and can be utilized in a wider variety of attacks and applications.

In our work, we strive to push the capacity of existing CPU side-channel attacks and apply them for novel attacks and applications, and in the meanwhile discovering new research aspects. More specifically, 1) we propose the concept of a prime+probe attack to extract onscreen keyboard inputs on Android, 2) we design and implement an automated approach to augment prime+pro be attack in the environment of aggressive cache prefetching and demonstrate significant improvement over traditional prime+probe attack, 3) we design a machine-learning-based system to automatically discover execution timing side-channels in graphics rendering libraries and using flush+reload attack to exploit them

on multiple platforms, evaluate using real-world applications and demonstrate its ability to infer sensitive user input with high accuracy, 4) we propose to use CPU side-channels as feedback to fuzzing when target binary cannot be modified and performed some initial evaluations, 5) we propose to use improve the coverage discovery rate of kernel fuzzing with reinforcement learning, implement it around Syzkaller [9] and significantly improve its coverage growth fuzzing Linux kernel. Ultimately, we demonstrate that CPU side-channels have great potentials and can be practically applied in many attacks and applications. Moreover, researching CPU side-channel attacks and applications can sometimes lead to interesting new research aspects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

CPU side-channels can be exploited to extract sensitive information using attacks such as *Flush+reload* [105] and *Prime+probe* [81]. These attacks can be launched by any user-space process and are able to bypass cross-process and even cross-VM boundaries. They have been widely used to extract secret keys from cryptographic algorithms including AES [94, 81, 62, 54] and El-Gamal [73]. Recently, the emergence of new CPU side-channels such as *Spectre* [65], *Meltdown* [71] and *Zombieload* [90] brought more much-needed attention to this security threat. The majority of researches on CPU side-channels focuses on extracting secret information from cryptographic algorithms, with several expectations such as [51]. However, the basic concepts of CPU side-channels are very general. Often times, the only requirement of CPU side-channel attacks is to have a co-residence malware with no special privilege. The same threat model that applies to existing cryptographic attacks can also apply to many other scenarios. As a result, there's bound to be a wider variety of possible attacks and applications utilizing CPU side-channels.

Launching a successful CPU side-channel attack is not a trivial task and faces the following challenges:

1. The capacity of such attacks are heavily impacted by CPU architecture, with features crucial to a successful attack (e.g. cache hierarchy, replacement policy) often not well-documented. Researchers put considerable efforts into understanding and reverse engineering these CPU features [58, 76, 25] but there are many other features that are not thoroughly studied yet.

2. Most CPU side-channels are timing side-channels, which means attackers can only infer the inner state of CPU by measuring the execution time of specific operations. Timing measurements are often subject to noise and can be affected by many external sources such as background processes, memory access speed, etc.

3. It is extremely challenging to discover vulnerable software code that is affected by known CPU side channels. Researchers often need to utilize their domain knowledge and manually analyze the program to identify these vulnerable codes. Traditional program vulnerability discovery approaches such as static/dynamic analysis are not designed to effectively discovering CPU side-channel vulnerabilities.

In this dissertation, we strive to push the capacity of existing CPU side-channel attacks, apply them for novel attacks and applications and address the challenges mentioned above. Specifically, we conduct the following studies:

1. We propose the concept of using a prime+probe attack to extract onscreen keyboard inputs on Android via its graphic buffer mechanism. We identified and studied several

challenges and sub-problems of this attack, one of which eventually leads to study #2. See Chapter 2 for more details.

2. We design and implement an automated approach to augment prime+probe attack in the environment of aggressive cache prefetching. We demonstrate our augmented prime+probe attack on real-world systems using the cache side-channel vulnerability (CSV) metric. We show that our approach doubles the information leakage compared to traditional prime and probe implementations. See Chapter 3 for more details.

3. We discover a novel type of execution timing side-channels in graphic rendering libraries. We design a machine-learning-based system to automatically discover these side-channels. We use flush+reload attack to exploit them on multiple platforms and predict users' sensitive text input with high accuracy. See Chapter 4 for more details.

4. We propose to use CPU side-channels as feedback to fuzzing when the target binary cannot be modified. We implement this idea around Syzkaller [9] kernel fuzzer and obtained some interesting observations, which sparks our idea of study #5. See Chapter 5 for more details.

5. We propose *SyzVegas* to dynamically choose the right fuzzing task in conjunction with the right seed, in Syzkaller [9]. Towards this, we model the specific fuzzing tasks as a multi-armed-bandit problem, which allows the system to learn the effective strategies and adapt over time, using a novel, yet intuitive reward assessment model to capture benefits and costs. See Chapter 6 for more details.

1.1 Background

1.1.1 CPU Caches

Programs often have temporal and spatial locality, i.e., the most recently accessed memory addresses, as well as nearby addresses, are often accessed in the near future. To exploit locality, modern architectures use CPU caches to store recently accessed memory. A CPU cache is often organized into multiple levels with different sizes and access speeds. For example, on Intel CPUs, there are commonly three levels of caches: L1, L2 and L3, with L1 being the fastest and the smallest and L3 being the largest and the slowest. On multi-core CPUs, lower levels of caches such as L2 and L3 are often shared among multiple CPU cores.

Modern CPU caches are organized using a *set-associative* policy. This policy divides the cache into multiple cache sets and each cache set contains several cache lines. When the CPU accesses memory, the memory address is indexed into a cache set. The CPU checks all cache lines in this set to identify the presence of the cache line holding the memory address.

Inclusiveness. Lower levels of the cache (L2, L3) can be configured with different inclusion policies. The most common policies are inclusive, exclusive, "non-inclusive non-exclusive" (NINE). Lower levels of the cache are considered inclusive if all the memory blocks present in the upper levels of the cache are also present on the lower levels. The lower levels of the cache are considered exclusive of the higher levels of cache if all the memory blocks present on the higher level of cache are not present on the lower level of cache. Otherwise. it is considered NINE.

Prefetching. Programs often access their memory in a predictable order. For example, when memory accesses and loads the predicted contents into the CPU cache before they are actually accessed. For example, Intel CPUs implement a streaming prefetcher that could prefetch up to 20 cache lines ahead of the cache line currently being accessed [6].

1.1.2 Shared Libraries

A program library is a collection of subroutines that are available for immediate use by other programs. Since the functionalities provided by these libraries are very commonly required, they are designed to be shared across multiple user programs. These libraries can be mapped to the address space of a user program by the linker when the user program prepares for execution. A library can also be loaded in the middle of an execution of a user program when it explicitly requests that the library be loaded. Regardless of the case, the contents in the library are mapped into the user program's address space.

Operating systems use shared memory to improve the memory utilization efficiency with regards to these libraries. For example, common libraries (`.so` on Linux, `.dll` on Windows) are often shared across all processes linking them. This means that these libraries are loaded into physical memory only once and remain there for the entirety of the OS session. For every process which loads a library, the library will be mapped to a different virtual address in the memory space of the process. However, when different processes access the same library, the same physical memory pages will be accessed.

1.1.3 Prime+probe Side-channel Attack

When a process loads a memory block from the cache, the access time is relatively short. If the accessed memory block is not present in the CPU cache, the process will need to load that block from memory (or a lower level of the cache), which is slower. This creates a timing side-channel for attackers to infer the current state of CPU cache, and thus perform attacks on the victim process. One of such attack is *Prime+probe* [81].

The prime+probe attack relies on the cache indexing and the replacement policy being known or discoverable. First, the attacker occupies (primes) all cache lines in a specific cache set with a set of memory blocks. If the victim process accesses memory that is indexed into the same cache set, one or more cache lines that the attacker previously occupied will be evicted based on the cache replacement policy. As a consequence, the attacker can then read (probe) the memory blocks previously occupying the cache set, check whether an eviction just happened and thereby infer the victim's activity. The prime+probe attack can work even when there is no shared memory between the attacker and the victim processes. However, since the size of the CPU cache is usually much smaller than the memory. thousands of different memory blocks can be mapped to the same cache set. This makes the prime+probe attack very noisy, especially on devices where many processes are running at the same time.

1.1.4 Flush+reload Side-channel Attack

Flush+reload [105] attack usually targets shared libraries. The attacker first picks a memory block in the shared libraries and flushes it out of the CPU cache. On Intel CPUs,

this can be done using the CLFLUSH instruction. If the victim process executes code that resides in the same memory block, it will load the memory block back into the CPU cache. Subsequently, the attacker accesses (reloads) the memory block, and checks whether it was loaded back into the CPU cache and thereby infers the victim's activity.

The flush+reload attacker does not have to consider issues of multiple memory blocks mapping on to the same cache set. Only the access of the exact memory block will produce a cache hit with the attacker's reload. This property makes the flush+reload attack substantially more precise than the prime+probe attack. In addition, the flush instruction supported by most architectures (e.g., CLFLUSH for Intel CPUs) is faster than executing sufficient memory accesses to occupy an entire cache set (needed by the prime), making flush+reload both more accurate and faster than prime+probe. The only drawback is that the flush+reload attack requires shared memory between the attacker and the victim but this property holds in our scenario.

On some CPU architectures such as ARM, cache flush instruction is not available to user programs. However, an attacker can still "evict" a cache line by accessing a set of memory blocks that are mapped to the same cache set. Gruss et.al. [72] demonstrated that evict+reload attack can be as effective as flush+reload.

1.1.5 Fuzzing

Fuzzing, or *fuzz testing*, is an automated software testing technique to discover software vulnerabilities. The goal of a *fuzzer* provides unexpected, or random data as inputs to the tested compute program to achieve a higher degree of code coverage and/or reveal bugs. In order to create valid input, fuzzers can apply either *generation* or *mutation*.

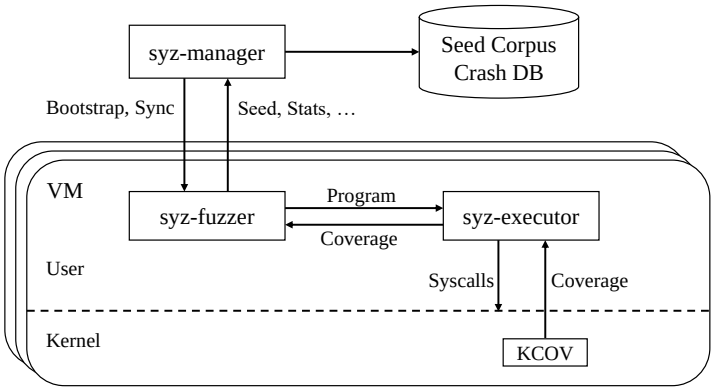


Figure 1.1: Architecture of Syzkaller.

A generation-based fuzzer (e.g. [43]) takes the well-defined input model provided by the user to generate inputs from scratch. The efficiency of such fuzzers are affected by the quality of input model. A mutation-based fuzzer (e.g. [18]) leverage existing valid inputs (aka. *corpus seeds*) and generates inputs by modifying (mutating) the valid input. Some fuzzers such as [9, 86] are capable of utilizing both strategies.

Many state-of-art fuzzers such as AFL [18], libFuzzer [22] and Syzkaller [9] often utilizes *gray-box fuzzing* or *coverage-guided fuzzing*. A gray-box fuzzer utilizes instrumentation to obtain information of the internal structures of the program. Fuzzer use these information to guide its fuzzing strategy in order to achieve better code coverage. Comparing to blind *black-box fuzzing* and program analysis-based *white-box fuzzing*, gray-box fuzzing has a reasonable performance overhead while informing the fuzzer about the code coverage during fuzzing, making it very efficient in detecting software vulnerabilities.

1.1.6 Syzkaller

Created by Google, Syzkaller [9] is the state-of-art kernel fuzzer. It is initially developed for fuzzing Linux kernel but later extends to support other OS kernels as well. Syzkaller is a grey-box fuzzer that relies on the kernel to provide code coverage information. Despite this, it can also be configured to run under black-box fuzzing mode. Syzkaller supports both generation and mutation. It comes with a comprehensive system call template including information on argument types and inter-system-call dependencies.

Figure 1.1 shows the architecture of Syzkaller. There are three key components: `syz-manager`, `syz-fuzzer` and `syz-executor`. `syz-manager` runs on the host machine and serves as the supervisor of the entire fuzzing process. It launches and manages guest VMs/devices, `syz-fuzzer` processes, and maintains the seed corpus and crashes database. If there are multiple fuzzer VMs/devices, `syz-manager` provides some degree of synchronization between them.

`syz-fuzzer` runs on the VM/device to be fuzzed. It handles the fuzzing logic (e.g. generation, mutation), creating programs (a sequence of system calls) and launch `syz-executor` processes to test the program. Once launched, `syz-executor` executes the program and interact with the code coverage component (e.g. *kcov* [20] for Linux) of the OS kernel, sending observed coverage back to `syz-fuzzer`. On Linux, the coverage is measured in number of edges between basic blocks. `syz-fuzzer` then analyze the coverage, comparing it to existing coverage. If new coverage is achieved, `syz-fuzzer` proceeds to analyze the program, attempts to create a seed program, and send seed program back to `syz-manager`.

1.1.7 Multi-armed Bandit Problem

Multi-armed Bandit (MAB) problem is a classic reinforcement learning problem. In this problem, a gambler must play a number of competing slot machine arms (choices) in a way that maximizes their expected gain. Each arm's properties are only partially known at the time of playing, and may become better understood as the arm is played more. The MAB problem is a classic example of the tradeoff between exploration and exploitation.

One notable variant of MAB problem is the is called the *Adversarial Bandit*, first introduced by Auer and Cesa-Bianchi in 1995 [27]. In this variant, the slot-machine arms are controlled by an adversary who is capable of altering the reward of each arm during every play. As one of the strongest generalizations of the MAB problem, adversarial bandit problem requires its solution to react quickly to the changing rewards of each arm.

1.2 Related Work

Prime+Probe: There has been an abundance of existing work on prime+probe CPU cache side-channel attacks. The Advanced Encryption Standard (AES) is the first to fall victim to prime and probe attack [81, 94, 62] where attackers were able to recover victim's memory accesses of AES lookup table, inferring the secret key. Prime and probe attack is also used to break other mechanisms such as El-Gamal [73]. Zhang et al. [109, 108] show that prime and probe attacks can even cross VM boundaries and perform cross-tenant attacks on PaaS (Platform as a Service) clouds. Moreover, prime+probe attack is shown to work not only on Intel CPUs but also other architectures such as ARM [70] and even well-sandboxed environments like browsers [78].

Flush+Reload: Flush+reload attack was first introduced by Yuval et al. [105]. In their work, the authors demonstrate that the flush+reload attack can be used to attack encryption applications such as GnuPG. Gulmezoglu et al., [54] design and showcase an improved flush+reload attack on AES. On ARM CPUs and Intel CPUs with non-inclusive LLC, there have been solutions that primarily leverage the cache coherency protocol among different last-level caches. For example, it was recently shown that the latest non-inclusive last-level cache employed by x86 CPUs can also be attacked [104]. Several researchers demonstrate the possibility of performing cache side-channel attacks on ARM. Zhang et al., [107] design and implement a return-oriented flush+reload attack on ARM. Both works utilize the cache coherency policy to monitor victim applications' instruction cache access and we adopted a similar methodology. Gruss et al., [72] perform a systematic study on cache side-channel attacks on the ARM architecture, discussing both the prime+probe and the flush+reload attacks.

Other CPU side-channel attacks: Recently, researchers are paying more attention to the implications of CPU optimizations (e.g. prefetcher, branch predictor, etc) on side-channel attacks. Gruss et al. [48] show that prefetch instructions from Intel and ARM CPUs can be utilized to infer sensitive information and break KASLR. Evtyushkin et al. [39] and Wang et al. [101] show that branch predictors can be exploited to attack secure systems such as SGX. The most well-known works in this area are Spectre [65] and Meltdown [71] attacks that utilize CPU's speculative execution feature to steal sensitive information from the kernel or other processes. Shin et al. [91] show that CPU cache stride prefetching introduces a side-channel that can be exploited against the ECDH algorithm in OpenSSL.

Effect of prefetching: There have been a few studies on the prefetcher's effect on cache side-channel attacks. Tromer et al. [94] is the first to acknowledge CPU prefetching's interference on prime and probe attack and propose a linked-list structure of eviction set and pointer-chasing technique to suppress the prefetcher. This technique is widely adopted in almost all known prime and probe implementations. Fuchs et al. [42] demonstrate that it is possible to defend prime and probe attacks by applying disruptive prefetching techniques to obfuscate victim's memory footprint.

Automated side-channel discovery: There are studies on the automated discovery of cache side-channels. Gruss et al. [51] proposes a cache template attack which aims to discover input-dependent cache line accesses automatically in shared libraries. Gorka et al., [59] utilize dynamic taint analysis to locate cache side-channels in crypto libraries. Wang et al., [100] model the cache behavior and use symbolic execution in conjunction with their model to discover crypto-related vulnerabilities.

Applying reinforcement learning to fuzzing: Researchers have attempted to apply reinforcement learning techniques to improving fuzzing efficiency. Woo et al. [103] use MAB algorithm to perform crash-based seed selection for black-box fuzzing within a fixed run/time budget. Patil et al. [84] treat the problem of assigning rewards the fuzzing iterations to a test case as a "Contextual Bandit" problem. Böttinger et al.,[31] use Q-learning [102] to learn a policy for choosing mutation operators. Karamcheti et al. [61] apply what is called a Thompson Sampling, bandit-based optimization approach to fine-tune the mutation operator distribution. MOpt [74] utilized a customized Particle Swarm Optimization (PSO) algorithm to determine the optimal distribution of mutation operators.

Seed selection optimizations for fuzzing: Researchers have also focused on optimizing different mutation strategies. Rebert et al. [87] explored several different seed selection algorithms and measured their qualities using linear programming. DigFuzz [110] uses a Monte-Carlo-based algorithm to prioritize favorable paths, and determine the seeds that are more valuable for future mutations. AFLFast [30] treats fuzzing as a Markov chain problem, modeling the probability that fuzzing the seed that exercises one path, generates an input that exercises another path.

Kernel fuzzing: Syzkaller provides a great foundation for coverage-based kernel fuzzing. There are several works built upon it, to improve kernel fuzzing from different aspects. FastSyzkaller [69] combines Syzkaller with an N-Gram model, to optimize the test case generation process. Moonshine [83] tries to improve the quality of the initial seeds in Syzkaller by "distilling" seeds from system call traces of real-world programs. RAZZER [60] combines fuzzer and static analysis, to detect race bugs in kernel. Difuze [38] utilizes static analysis to compose correctly-structured inputs in the userspace, to explore kernel drivers.

Besides Syzkaller, there exist other fuzzers for OS kernels. Trinity [24], iknowthis [19], KernelFuzzer [21], and sysfuzz [23] are built with hard-coded rules and grammars. kAFL [88], TriforceLinuxSyscallFuzzer [47], TriforceAFL [57], on the other hand, are based on or inspired by AFL.

Chapter 2

Prime+Probe Attack Against Android Graphic Buffers

This chapter documents our research on using prime+probe attack against Android graphic buffers.

2.1 Introduction & Intuition

In graphic rendering, it's common pratice that graphic buffers be maintained to store rendered graphics before pushing them to the display. For graphic architectures such as Linux X Window System [12] and Microsoft Windows, such buffers are maintained by a system process or the kernel itself. Other platforms such as Linux Wayland [11] and Android lets the applications maintain its own buffers while system services are dedicated to compositing graphic buffers from multiple applications.

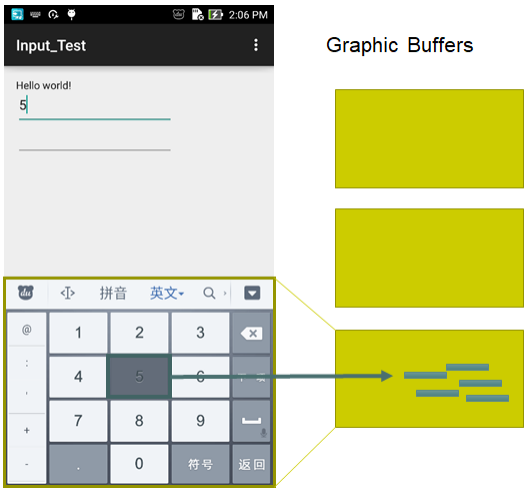


Figure 2.1: Android Graphic Buffer and Side-channel

Figure 2.1 demonstrates a example of Android graphic buffer if an Input Method Editor (IME) application (aka. onscreen keyboard). The process maintains three graphic buffers in its memory and alternating them when an UI change occurs. When part of the UI need to be updated, the process will update the modified part of its current graphic buffer, submit the buffer to the dedicated system graphic service and setting the next graphic buffer as the current buffer.

It is worth noting that for IME applications, different user input often result in different part of the UI being updated. This creates a side-channel for a local attacker to exploit. If the attacker can monitor what part of the graphic buffer is being updated, attack can infer user's sensitive input.

As a result we propose to use prime+probe attack to extract user's sensitive input from graphic buffers of IME application, as shown in Figure 2.2. As demonstrated by previous research [81, 94, 62], prime+probe attack is effective in monitoring what part of

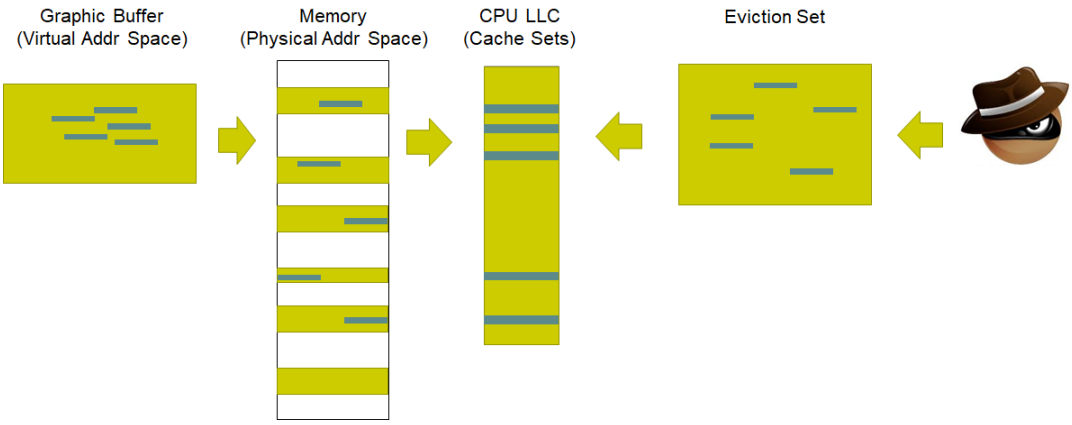


Figure 2.2: Prime+probe attack on Android graphics buffer

the CPU cache has been accessed by the victim. When user presses a key on IME process, the IME process will often render a highlight effect of the pressed key. During this process, the IME process will make a partial update to its graphic buffer. Such update will eventually be made to the physical memory containing the graphic buffer, as well as the corresponding cache sets in the CPU's unified last-level cache (LLC). Different keypress will result in different part of the graphic buffer being modified, thus touching different subsets of CPU LLC. As a result, the attacker monitoring the CPU LLC can observer what part of the LLC is being touched by the IME app, thus inferring user's input.

In order to implement an end-to-end attack, there're several technical challenges that must be addressed:

1. **Prime+probe entire LLC in a single run.** Graphic rendering is triggered by the victim user and happens only once per keypress. Thus it is essential that the attacker be able to monitor the entire CPU LLC during the process, unlike previous researches (e.g. [108]) that relies on repeatedly launching victim for a large number of times.

This is not straightforward as prime+probe is impacted by the CPU data prefetcher. We manage to address this technical challenge in our research in Chapter 3.

2. **Construct physical mapping.** CPU LLC is often physically indexed. In order to perform the attack, it's essential that a unprivileged attacker knows the virtual-to-physical mapping of the IME process's graphic buffer. On Asus Zenfone 2, we managed to reverse-engineer its graphic buffer allocation driver and trick it into allocating physical memory space designated by the attacker. We also propose to use prime+probe attack in combination with the CFS scheduler attack [52] to infer the physical address of the graphic buffers during their initialization.

3. **Slowing down the rendering.** The rendering time of highlighted key can range from 0.01 to 0.35 microseconds. However, occupying and probing 1-2 MB of CPU LLC takes much longer. In addition, the highlighted key-press itself will often result in a large ($> 70\%$ in our experiment) proportion of the CPU LLC being accessed, making it harder to differentiate between different key presses, especially in the presence of noise. Therefore, it is necessary to use CFS scheduler attack [52] to slowdown the rendering process, allowing the attacker to collect multiple measurements.

2.2 Conclusion

According to previous discussions, it is essential to have the ability to slowdown the victim IME app via CFS scheduler attack. However, Android gives foreground apps and background apps different scheduling priorities. The CFS scheduler attack can only work on the IME app (which is a foreground app when being used) when the attacker's app is

also a foreground app. The only way to achieve this is through the `SYSTEM_ALERT_WINDOW` permission, which allows a background app (e.g. messenger app) to draw graphic content (e.g. new message notification) on top of other foreground apps, thus granting it the same scheduling priority. Unfortunately, Fratantonio et.al [40] has demonstrated that with this permission, there is a much easier and more reliable approach to steal user's sensitive input from IME apps, making our proposed attack obsolete. Despite this setback, we are still intrigued by our research into this potential attack. Furthermore, this attack gives us the idea of improving traditional prime+probe attack by allowing it to monitor the entire CPU LLC at once, which we will introduce in Chapter 3. Finally, on to the topic of attacking users' sensitive input, we designed and implemented an end-to-end CPU side-channel attack from a different angle, which we will introduce in Chapter 4.

Chapter 3

PAPP: Prefetcher-Aware Prime+Probe Attack

This chapter documents our research into improving classic CPU cache prime+probe attacks in the presence of prefetching, which is published in Design Automation Conference (DAC) 2019. [99]

3.1 Introduction

CPU cache side-channels can be exploited to extract sensitive information [105, 81]. These attacks can be launched by any userspace process and are able to bypass cross-process and even cross-VM boundaries. The most general of these attacks is Prime and Probe which has been widely used to extract secret keys from crypto algorithms including AES [94, 81, 62] and El-Gamal [73]. To successfully implement a prime and probe attack, researchers put considerable efforts into understanding and reverse engineering CPU features [58, 76, 25]

One aspect of side-channel attack that has not been well studied is the impact of the data prefetcher, which speculatively fetches un-accessed cache lines to improve cache hit rate. The prefetcher adds noise to the side-channel information in two ways: first, the victim signal has some spurious accesses that are from the prefetcher rather than the application. Moreover, the attacker's prime and probe access patterns are limited since it also generates unneeded memory accesses from the prefetcher. The effects described above can substantially interfere with prime and probe attacks. One commonly adopted methodology is to utilize a linked-list setup of the eviction set [94] to suppress prefetching. By adopting this approach, an attacker can suppress the prefetcher to the next-line prefetcher only and thus enable prime and probe of every other cache set. Still, this approach leaves the attacker missing half of the cache sets, lowering the quality of the leaked signal. To make things worse, on CPUs with more aggressive prefetching, the attacker might not even be able to prime and probe every other set, making the attack way less effective. In particular, in-order processors such as the Intel Atom, which is often used in embedded systems, have very aggressive prefetchers since cache misses cause substantial performance losses without the support of out-of-order execution.

In this paper, we develop a new prime and probe attack which we call Prefetcher-Aware Prime and Probe (PAPP). PAPP reverse-engineers the replacement policy and the prefetching behavior and generate a prime and probe pattern in an automated fashion. PAPP mitigates the impact of the prefetcher on prime and probe attacks, substantially improving their effectiveness (especially in the presence of aggressive prefetchers). Our experiments show that PAPP can almost completely circumvent the effect of cache prefetching

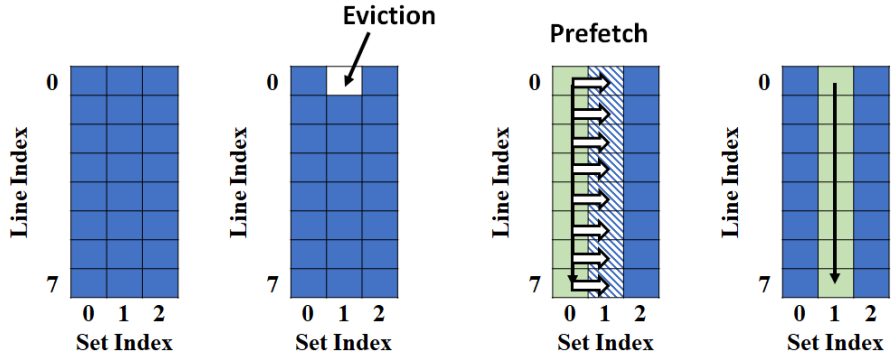
on in-order CPUs with aggressive prefetching policy, substantially improving the quality of prime and probe attack.

Our main contributions are:

- We perform a systematic study on the impact of prefetching on prime and probe attacks. We demonstrate limitations of the existing implementation of prime and probe attack.
- We present a novel prime and probe strategy aiming to address the effect of prefetching. We combine the knowledge of replacement policy and cache prefetcher to effectively circumvent the effect of cache prefetching. We automate the generation of prime and probe strategy and open-source our implementation at [4].
- We evaluated PAPP on real-world system using cache side-channel vulnerability (CSV) metric. We show that our approach doubles the information leakage comparing to traditional prime and probe implementations.

3.2 Background and Motivation

The high cost of memory accesses is one of the fundamental bottlenecks limiting processor performance. Processors use caches (often *Set-associative*) to store recently accessed memory in order to compensate this cost. Moreover, modern CPUs exploit predictable program access patterns using prefetchers that preload memory that is likely to be accessed in the future [95]. Prefetchers can be quite aggressive, especially in in-order processors which are often used in embedded applications: in such processors, cache misses cannot



(a) Initial state (b) Victim's eviction (c) Signal erased (d) Cache hit

Figure 3.1: Prefetcher's effect on traditional prime+probe

be compensated for using out-of-order execution. For example, Intel CPUs implement a streaming prefetcher which could prefetch up to 20 cache lines ahead [6].

Because CPU caches are shared among multiple programs, they become targets of side-channel attacks [81]. Prime+probe is one of the most general attack strategies because it does not require shared memory pages with the victim. At high level, the attacker starts with completely filling (*prime*) the cache sets she wish to monitor using a carefully chosen *eviction set*. When the victim generates memory references, its accesses replace some of the cache lines in the eviction set filled by the attacker. The attacker can then access the eviction set again (*probe*); whenever an access results in a cache miss, she can infer that the victim has accessed that cache set resulting in her data being replaced.

The most commonly used approach to implement a prime+probe attack is to sequentially access all cache lines in the same cache set from the eviction set, timing the total access time [81, 73]. This way, priming and probing are built into one access pattern, making it efficient to monitor a single cache set.

Unfortunately, this approach does not work well with multiple cache sets in the presence of a prefetcher. Figure 3.1 shows an example of a prime+probe attack on 8-way set-associative cache. First, the attacker primes all cache lines in sets 0,1,2 and waits for victim's activity (a). Next, victim's memory activity evicts cache set 1 (b). In (c), the attacker starts to prime+probe cache set 0, getting all cache hits due to the lack of victim's activity on this set. However, during this process, the prefetcher loads all cache lines from cache set 1 back, completely erasing the signal created by the victim (d).

This issue cannot be solved by simply going backward in prime+probe order. On many modern CPUs, the prefetcher can also detect whether an attacker accessing memory in forward order or backward order, changing prefetching tactics accordingly. Tromer et. al. [94] attempted to set up a random-order linked-list structure and utilize a pointer-chasing technique when accessing eviction set memories, suppressing the stream prefetcher from aggressively loading too many cache sets because of the unpredictable access pattern. However, we find that this approach still cannot completely get around the next-line prefetching (a standard prefetcher that always brings in the next cache line), especially on in-order CPUs with more aggressive prefetching. As a result, attackers often compensate for this prefetching issue by either skipping every other cache set and/or repeatedly conducting experiments and testing different cache sets each time [109]. In either case, the performance and precision of prime+probe is severely impacted, potentially making it unusable for CPUs with aggressive prefetchers.

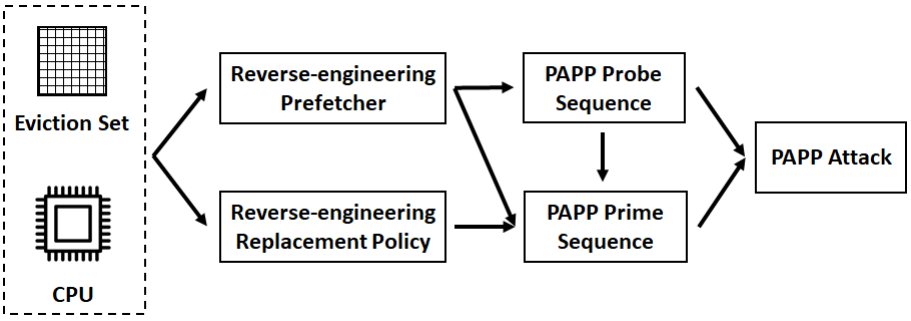


Figure 3.2: PAPP Attack Workflow

3.3 PAPP Design and Implementation

We propose Prefetcher-Aware Prime and Probe (PAPP) attack, a prime+probe attack that overcomes the negative impacts of prefetching. Figure 3.2 demonstrates a high-level workflow of the attack. Similar to other prime+probe attacks, the attacker first creates an **eviction set**. Utilizing this eviction set, PAPP first conducts **prefetcher reverse engineering** and **replacement policy reverse engineering**. Using the obtained profiles of the prefetcher and replacement policy, PAPP then constructs a **probe sequence** and subsequently a **prime sequence**. PAPP then combines both sequences into a prime+probe attack that circumvents the interference of the prefetcher.

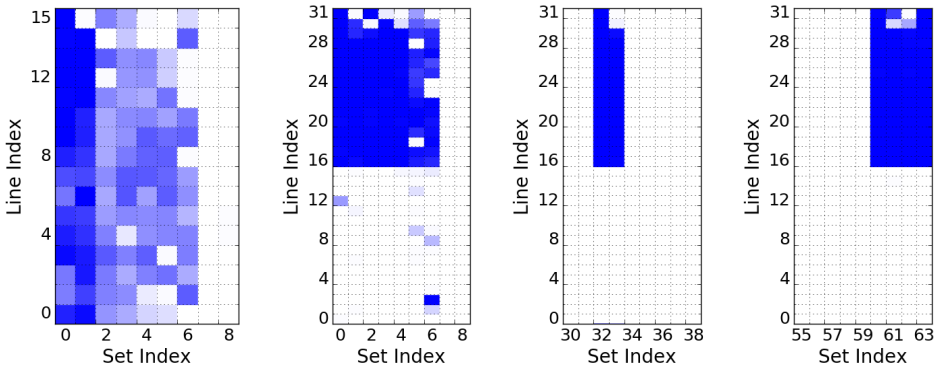
Specifically, PAPP leverages two new ideas: (1) It first reverse-engineers the replacement policy to make the probe sequence possible using only one access to each set (instead of having to access all the cache lines in each set). As a result, fewer accesses are generated, leading to less prefetching activities; and (2) It uses probe patterns that avoid the impact of the prefetcher. Together, the techniques allow for near perfect probing of the cache, even on in-order CPUs with aggressive prefetching. Although we omit the details due to space, we are able to automate the construction of these sequences giving our profiling,

potentially enabling the attack to be adaptable with little effort to other CPUs. Detailed implementation of algorithms used by PAPP can be found at [4].

To simplify explanations, and without loss of generality, we assume that the attacker targets a single memory page. We design and implement PAPP attack on Intel Atom Z3560 and Z3580 and use the L2 cache of Z3580 for demonstration. Intel Atom is an in-order processor with a unified 16-way set-associative L2 cache. It has a cache line size of 64 bytes, which means it would require 64 cache sets to cover a 4KB memory page. We selected the Atom, as an in-order processor representative of what is used in embedded systems, because such processors are known to use aggressive prefetchers. All results are from real experiments conducted on an Android phone using Atom Z3580 CPU.

3.3.1 Reverse Engineering the Prefetcher and Replacement Policy

Eviction Set: We construct the eviction set ES with m sets and n lines per set similar to other prime+probe attacks (e.g., [62]). Let a be the associativity of the CPU cache level being targeted, i.e. each cache set is composed of a cache lines. Therefore, in order to fully occupy the cache, we need $n \geq a$. We partition the eviction set ES in to two sections: **occupation section** and **warmup section** (which is a new improvement we introduce here). The occupation section is composed of $m \times a$ cache lines, designed to occupy the cache after priming. This section is essential to all prime+probe attacks. On CPUs with strict least-recently-used (LRU) replacement policy, the warmup section is not needed. However, most modern CPUs do not have a strict LRU policy [25], the warmup section is a necessity which is used to make the occupancy and replacement state of the cache set more predictable as we demonstrate later in the paper.



(a) n=a, s=0 (b) n=a+16, s=0 (c) n=a+16, s=32 (d) n=a+16, s=63

Figure 3.3: Occupancy on L2 cache of Atom Z3580. Darker cells means higher chance of cache line being in cache. Each cache line tested 100 times.

Reverse Engineering Prefetcher: We reverse engineer the CPU memory prefetcher and use the results later in Section 3.3.2 to construct PAPP prime+probe sequence. Prefetcher reverse engineering aims to answer the following two questions:

1. What cache lines will be occupying the CPU cache after accessing a sequence of memories in the eviction set?
2. What cache lines will be first replaced upon victim's access to the same cache set?

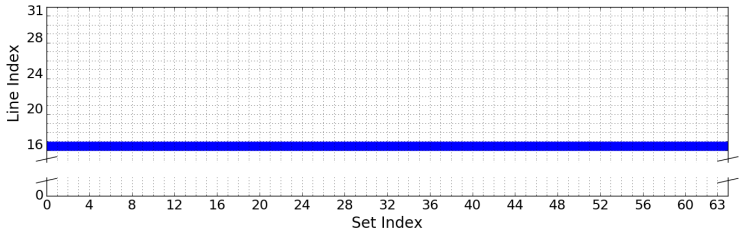
We perform prefetcher reverse-engineering using two steps: 1) Access a sequence of cache lines in the eviction set ES and 2) Access an arbitrary cache line in ES and check whether the this line is cached (indicating it was loaded or prefetched). We adopt the pointer-chasing technique used in previous literature [94] during the accesses suppress the prefetcher.

The prefetcher behavior is complex and can vary based on the access pattern and the availability of memory bandwidth. To provide a basic characterization, we conduct prefetcher reverse-engineering with respect to accesses to a single set, i.e. what cache lines will be in the cache after we fully prime a single set s in by accessing all cache lines mapped to s in the eviction set. The result on Intel Atom Z3580 processor is shown in Figure 3.3.

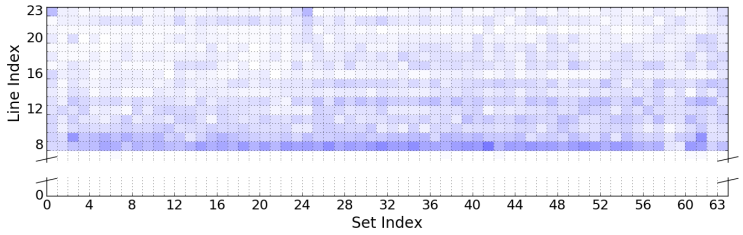
Figure 3.3(a) demonstrate the cache occupancy heat map of the eviction set after priming set $s = 0$ in the eviction set without any warmup section (i.e. $n = a = 16$). We notice that Atom does not have a naive LRU replacement policy, as accessing 16 cache lines in the eviction set does not guarantee full cache occupancy of the accessed lines. As a result, we determine that a warmup section is necessary for reliable cache priming.

Figures 3.3(b), 3.3(c), 3.3(d) shows the cache occupancy heat map after priming set 0, 32 and 63 respectively with 16 extra lines as warmup (i.e. $n = a + 16 = 32$). First, we see that using 16 extra lines reliably ensures the cache occupancy of the occupancy section of the eviction set (lines 16-31). Additionally, we found that the prefetching behavior differs for different cache sets. The prefetcher is more aggressive at the beginning of the page (cache set 0) than at the middle of the page (cache set 32). And towards the end of the page (cache set 63) the prefetcher will prefetch previous cache set instead of next cache set apparently to avoid prefetching into a potentially unmapped or uncached page.

Based on the analysis above, we conclude that although traditional prime+probe will be able to monitor every other set at the middle of the memory page, it's impossible to do so at the beginning and end of the page. The effectiveness of traditional prime+probe is substantially impacted by aggressive prefetching.



(a) $n = a + 16$



(b) $n = a + 8$

Figure 3.4: Replacement profiling on L2 cache of Atom Z3580. Darker means higher chance of line being replaced first. Each cache line tested 100 times.

Reverse Engineering Replacement policy: We reverse engineer the replacement policy which is needed to construct the prime+probe sequences. If we can reliably set up which cache line will be replaced if a set is accessed by the victim, we can probe only this line to determine if there is a victim access, substantially reducing the number of memory accesses and prefetcher noise.

Similar to prefetcher profiling conducted in Figure 3.3, we conduct replacement profiling by first fully priming a cache set s. Afterwards, for each cache line in s, we compare the miss rate of this cache line with/without a single victim memory access in set s as an indicator of how often a cache line in the eviction set will be replaced after a single victim memory access at the same cache set. Figure 3.4 shows our profiling result on Intel Atom

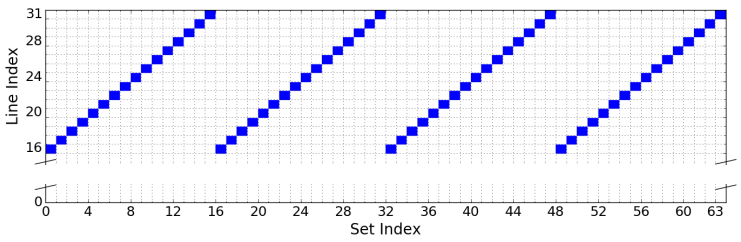


Figure 3.5: Sample probe sequence for L2 cache of Intel Atom.

Z3580. According to Figure 3.4(a), when we have a 16-cache-line warmup section, the 16th least recently used cache line reliably becomes the next victim. However, as we can see in Figure 3.4(b), with a warmup section size of 8, the replacement status becomes much less predictable.

3.3.2 PAPP Prime and probe Sequence

With the knowledge of the prefetching and replacement policies, PAPP crafts a prefetcher aware prime+probe sequence.

Probe Sequence: To avoid prefetcher effects, accesses in the probe sequence should not prefetch memory from the rest of the sequence; otherwise prefetched data overwrites any victim data. Moreover, accessing probe sequence should not prefetch new data into the cache; otherwise prefetched data evicts attacker's eviction set data. We have automated the construction of probe sequence using the prefetcher reverse-engineering result we discussed in Section 3.3.1. We start with an empty sequence $seq_{probe} = ()$. For each cache set s in the eviction set ES, we try to find a cache line indexed to s in the occupation section of ES (thus prevents self-evicting) such that accessing seq_{probe} does not prefetch the chosen cache line. Upon finding such cache line, we append it to seq_{probe} and move on to the next

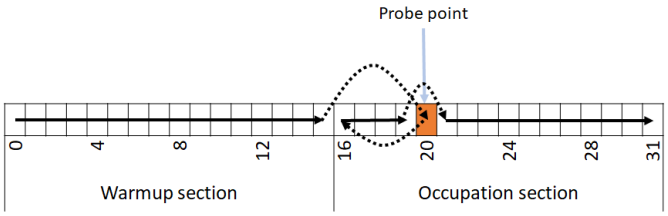


Figure 3.6: Sample prime sequence (solid line) for L2 cache of Intel Atom which sets line 20 to be replaced next.

set until all sets are covered. There can be multiple possible probe sequences that satisfy our requirements. Figure 3.5 illustrates one possible generated probe sequence for Atom Z3580's L2 cache.

Prime Sequence: For each cache set s, we seek a memory access sequence of cache lines in ES that can (1) occupy the cache with the occupancy section in ES and (2) set the chosen cache line in the probe sequence to be the next one being replaced. We automate this process utilizing the reverse-engineering result of the prefetcher and replacement policy. Figure 3.6 shows a prime sequence of one set generated on Intel Atom Z3580. The sequence first access all cache lines in the warmup section. Next, it accesses line 20 and finally rest of cache lines in occupation section. According to our reverse-engineering result in Figure 3.4(a), this will reliably set line 20 as the next one being replaced.

Generating prime sequence will be problematic when there is forward prefetching and backward prefetching at the same time. In the case of Atom Z3580, priming cache set 62 will prefetch all cache lines in set 61 while priming cache set 61 will prefetch set 62. Therefore whichever set got primed first will have its replacement status wiped when priming the other set. Fortunately, for Atom Z3580, such backward prefetching only exists

when priming cache set 62 and 63. In practice, the attacker can omit these two cache sets in order to ensure other cache sets are monitored effectively.

3.4 Evaluation

We implement both PAPP and traditional prime+probe attacks in C and perform the attack on Intel Atom Z3580 CPU running Android OS. For traditional prime+probe attack, we use a standard implementation following previous attacks [62, 109, 52] which prime+probe every other cache set and uses pointer chasing to suppress the prefetcher. In contrast, PAPP is able to probe all sets with the exception of sets 62 and 63 as discussed earlier. We use a benchmark victim program and test the ability of prime+probe attack towards inferring victim's activity. We do not apply any prefetcher suppression techniques to the victim. We consider three victim process access patterns:

1. Accessing one cache line: in this pattern, the victim's behavior corresponds to LLC-based AES attacks introduced in [62, 109, 52]. In these attacks, the attacker exploits the Linux complete fair scheduler (CFS) to interrupt the victim's execution. As a result, the victim can only access a single AES table entry between two attacker prime+probe rounds.

2. Accessing six consecutive cache lines: corresponds to attacks on El-Gamal cipher [73], where each table entry spans 6 cache lines. CFS exploit is optional since the crypto computation is much slower than AES.

3. Accessing a various number of random cache lines: this case provides a tunable more general access pattern.

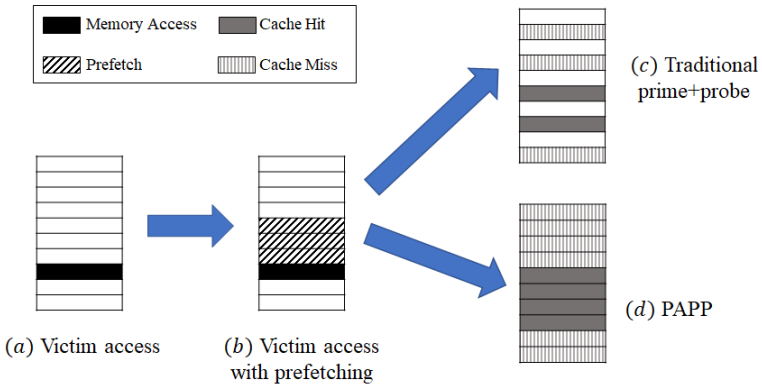


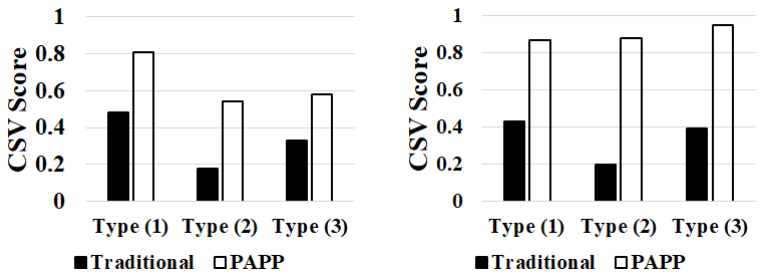
Figure 3.7: Victim's prefetching

To measure the effectiveness of the attack, we use the cache side-channel vulnerability (CSV) [106] metric. CSV computes Pearson correlation coefficient between the victim's cache activity (oracle) and attacker's measurement. The higher the CSV, the stronger the correlation between victim's activity and attacker's measurement, indicating better attack effectiveness.

3.4.1 Modification to CSV Metric

CSV does not account for CPU cache prefetching and has only been used in experiments where the prefetcher is disabled. The original work [106] assumed that the prefetcher can be either disabled or fully suppressed. As a result, it only considers the victim accesses from the application perspective rather than their footprint in the cache as observed by the attacker. In reality, however, the prefetcher affects victim's memory footprint and in turn affect attacker's observations.

Figure 3.7 demonstrates one attack scenario. In this example, the victim accesses one cache set while an attacker (traditional or PAPP) is trying to infer victim's access. To



(a) Next-line prefetch prediction. (b) Full prefetch prediction.

Figure 3.8: CSV score with prefetch prediction.

measure the effectiveness of the attacker, CSV computes the correlation between attacker's observation (Figure 3.7(c), Figure 3.7(d)) and victim's access (Figure 3.7(a)). With the presence of prefetcher, however, the attacker does not directly monitor victim's memory access. Instead, attacker can only monitor a combination of victim's memory access and victim's prefetched memory access (Figure 3.7(b)). Therefore, computing the correlation between (a),(c) and (a),(d) does not accurately reflect the success in recovering the cache state, as some of the attacker's observations can only correlate to the victim's prefetching.

To address this issue, for the victim access pattern we include the prefetching behavior based on a model of the prefetcher. A simple model is to assume that the prefetcher will only prefetch next cache line for the victim. A more sophisticated model can use the prefetcher profile (as we carried out in the previous section) to more accurately predict victim's prefetching behavior. Note that we only modify the computation of the CSV metric, not to the operation of PAPP or traditional prime+probe experiments.

3.4.2 Comparison to Traditional Prime+Probe

Figure 3.8 shows the CSV score with the traditional attack and PAPP. We found that for type (1) victim, the prefetcher indeed only prefetches the next cache line except at the beginning and end of a page. PAPP substantially outperforms traditional prime+probe across all cases: for example, for type(1) workload, PAPP achieve a CSV of 0.81 using the modified next line CSV metric (Figure 3.8(a)) and even higher with the full prefetch prediction (Figure 3.8(b)), while traditional prime+probe scores only 0.48, demonstrating that PAPP is a much higher quality attack. Since traditional prime+probe can only probe every other cache set, it cannot capture access to sets not being probed, resulting in a lower correlation.

For type (2) and type (3) victims, we found that the prefetcher is more aggressive when multiple closely-located cache lines are accessed. In type (2) victim, we found that the prefetcher is constantly prefetching 10-11 cache lines. In type (3) victim, the aggressiveness of the prefetcher depends on the memory access pattern of the victim. As a result, a naive next-line prediction can no longer reflect the cache activity of the victim program. This can be compensated by having a better profile of victim's prefetching behavior, as shown in Figure 3.8(b). We see that with the correct prefetcher profiling, the CSV for type (2) victim rises to 0.2 (traditional) and 0.88 (PAPP), while the CSV for type (3) victim rises to 0.39 (traditional) and 0.95 (PAPP). We notice that type (2) victim has a low CSV under traditional prime+probe. This is because the prefetcher is more aggressive for type (2) victim, prefetching cache lines in more sets that traditional prime+probe cannot monitor.

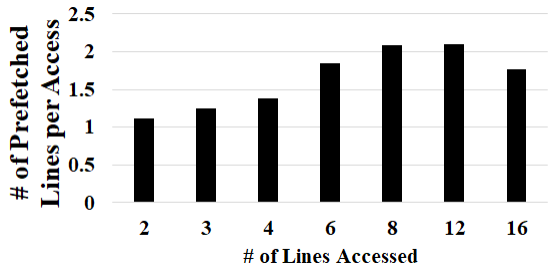


Figure 3.9: Prefetched cache lines for type (3) victim.

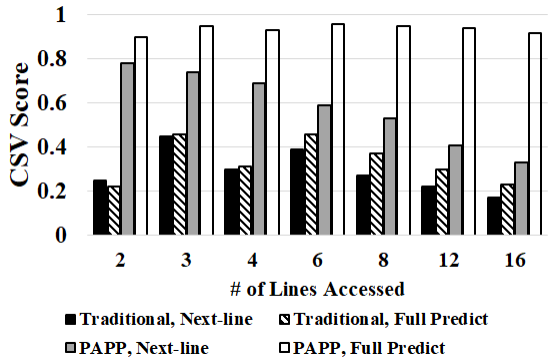


Figure 3.10: CSV for type (3) victim

Figure 3.9 further demonstrates the effect of the prefetcher when a type (3) victim accesses various numbers of cache sets. On average, the prefetcher prefetches no more than 2.1 additional lines per victim's memory access. Figure 3.10 shows the CSV score for type (3) victim with different number of accesses. We notice that as the intensity of the victim's activity increases, next-line prediction approach loses its effectiveness. A refined "full prediction" is necessary when victim accesses more memory during one round of prime+probe. As we mentioned in Section 3.4.1, this observation only refers to the computation of the CSV metric. The operations of PAPP or traditional prime+probe remain unmodified.

3.4.3 Discussion

PAPP exploits the replacement policy and attempts to set the cache replacement to a more predictable status. Through our experiment with Intel Atom processors, we show that this can be achieved on processors without naive LRU replacement policy. On ARM CPUs, however, we are unable to achieve the same since ARM's replacement policy is much less predictable as demonstrated in previous research. [70]

We also experimented on some out-of-order Intel CPU architectures including Xeon, Haswell, Sandybridge and Skylake. PAPP can successfully generate a prime and probe strategy on these CPUs but their prefetchers behaves very differently from Atom. On Xeon, we found that pointer chasing technique can effectively disable next-line prefetcher. On Haswell, Sandybridge and Skylake, we found that priming a cache set with odd index will prefetch the next line while priming a cache set with even index will prefetch the previous line. As we discussed in Section 3.3.2, the generated prime and probe strategy cannot monitor adjacent cache sets and achieve higher coverage than traditional prime and probe on these CPUs One conclusion of this is that PAPP is most important for in-order processors which have more aggressive prefetching.

Besides coverage, PAPP also has an advantage of manipulating the replacement policy and separating probe sequence with prime sequence. Specifically, when probing results in a cache hit, an attacker can simply skip this set during priming, improving the throughput of prime and probe drastically. The effect on replacement status can be compensated by updating the prime and probe sequence for the next round. This benefit applies to both CPUs with aggressive prefetchers and CPUs with less aggressive prefetchers.

We show that with the prefetcher-aware approach of PAPP, we can monitor more cache sets than traditional prime and probe attacks. We believe this makes PAPP applicable on a wider variety of attacks, especially in scenarios where attacker can only obtain a limited number of observations. (e.g. [98]) We plan to implement new cache side-channel attacks using PAPP in the future.

3.5 Related Work

There has been an abundance of existing work on prime and probe CPU cache side-channel attacks. The Advanced Encryption Standard (AES) is the first to fall victim to prime and probe attack [81, 94, 62] where attackers were able to recover victim's memory accesses of AES lookup table, inferring the secret key. Prime and probe attack is also used to break other mechanisms such as El-Gamal [73]. Zhang et al. [109, 108] show that prime and probe attacks can even cross VM boundaries and perform cross-tenant attacks on PaaS (Platform as a service) clouds. Moreover, prime and probe is shown to work not only on Intel CPUs but also other environments such as ARM [70] and browsers [78].

There have been a few studies on prefetcher's effect on cache side-channel attacks. Tromer et al. [94] is the first to acknowledge CPU prefetching's interference on prime and probe attack and propose a linked-list structure of eviction set and pointer-chasing technique to suppress the prefetcher. This technique is widely adopted in almost all known prime and probe implementations. Fuchs et al. [42] demonstrate that it is possible to defend prime and probe attacks by applying disruptive prefetching techniques to obfuscate victim's memory footprint. Unfortunately, to our knowledge, this technique is not implemented in any CPUs.

Recently, researchers are paying more attention to the implications of CPU optimizations (e.g. prefetcher, branch predictor, etc.) on side-channel attacks. Shin et al. [91] show that CPU cache stride prefetching introduces a side-channel that can be exploited against ECDH algorithm in OpenSSL. Other CPU optimizations such as branch predictor (e.g. [39]) and speculative execution (e.g. [65, 71] have also been exploited for side-channel attacks.

3.6 Conclusion

In this paper, we propose PAPP: a prefetcher-aware prime+probe cache side-channel attack. PAPP performs systematic reverse-engineering of CPU cache prefetcher and replacement policy. We show that PAPP is able to construct prime+probe strategy that are resistant to the interference of aggressive prefetchers on in-order CPUs. We evaluated PAPP on real-world system using cache side-channel vulnerability (CSV) metric and demonstrates that PAPP doubles the information leakage comparing to traditional prime+probe implementations. We hope that in the future, PAPP can be used in new attacks and applications to provide efficient cache monitoring.

Chapter 4

CPU Cache Side-Channel Discovery and Attack on Graphic Libraries

This chapter documents our research on automated discovery and exploitation of timing side-channels in graphic libraries, which is published in Network and Distributed System Security Symposium (NDSS) 2019 [98].

4.1 Introduction

Graphics are pervasively used in modern applications and many applications implement a graphical user interface (GUI) to improve user experience. Graphics rendering is complex and involves multiple processes. For example, on the Linux X architecture, graphics rendering involves many components spanning the kernel, the X-server, the application

client, and the device driver. To shelter developers from this complexity, many operating systems provide graphics libraries with simple APIs for applications to process and render their GUIs.

In this chapter, we scrutinize such graphics libraries as a target of side-channel attacks. It is noteworthy that these graphics libraries are provided by operating systems and loaded dynamically by applications and shared across user processes, i.e., different virtual pages are mapped to the same physical pages. This creates an opportunity for a malicious process to infer graphics-related activities of a victim process.

Our intuition of the attack is that the performance of graphics rendering is critical for user experience across a wide range of applications. Consequently, graphics libraries often optimize their execution logic for high performance. For example, when handling simpler graphical content, graphics libraries usually execute a different set of procedures than that for complex content. Even if the same set of subroutines are executed, the execution time can still differ for different inputs (e.g., different characters to render). This processing logic creates a side-channel that can allow attackers to infer a user's input since the execution times of these sensitive graphics operations are both input-dependent and measurable.

However, the practical realization of such side-channel attacks is not trivial, especially since graphics rendering is complex, large, and is characterized by interdependence across multiple processes. On Linux, for example, we find that graphics rendering could involve multiple shared libraries and millions of lines of code. Thus, if attempted blindly, it would take significant time and manual effort to determine where side-channels exist during

the rendering process. Moreover, even upon finding such a side-channel, it is difficult to assess whether the leakage is sufficient to reliably recover the target information (e.g., in the presence of measurement noise). To further complicate attacks, measuring the execution time of the victim process regarding the shared graphics library is also challenging. Previous attacks often rely on the attacker *actively triggering* the sensitive procedure (e.g., encryption), and measuring the execution time as many times as they want [66, 34]. When attacking graphics-based applications such as the ones we consider, the attack process can only *passively observe* the execution, leaving fewer attack opportunities.

To this end, we propose a novel method to completely automate the end-to-end realization of practical attacks on graphics rendering. It not only automatically identifies the vulnerable instructions/subroutines whose execution times are input-dependent, but also yields an end-to-end exploit to infer a user's inputs with disturbingly high accuracies. The method measures the information gains from a set of execution times of the subroutines involved in rendering, and identifies those that yield high information gain (i.e., allow effective discrimination between subroutine executions [55]). We then apply a machine learning model that uses these discriminatory subroutines' executions to infer the user's input with high accuracy. We demonstrate that the method exposes a leaky CPU side-channel that is practically exploitable and more effective than previously known side-channels of similar types (e.g., [51]). Unlike many previous studies (e.g., [29, 33, 54]), where the researchers relied on manual inspection of source code or instructions to identify vulnerabilities, we provide a systematic methodology that can completely automate the identification of such side-channels (all of which are previously unknown).

We upload the demo videos of our attack on an anonymous website [3]. Moreover, to facilitate the reproduction of the work and future research, we open source the complete source code of the attack at [4].

Our contributions: In brief, we make the following key contributions.

1. *Exposing input-dependent execution-time side-channels in graphics libraries:* We systematically investigate this unique type of under-scrutinized side-channels in graphics libraries. By developing a novel and automated methodology, we discover previously unknown and exploitable graphics rendering side-channels on both Ubuntu and Android platforms.

2. *Accurate subroutine execution time measurement:* We design and implement an end-to-end CPU cache side-channel attack to measure the execution time reliably in graphics rendering subroutines. We address the technical challenges associated with noises and the implementation of such attacks on the Android platform.

3. *Evaluations on real-world applications:* We demonstrate that the discovered side-channel on common graphics libraries can be exploited to infer the passwords with lowercase letters and numbers 10,000 - 1,000,000 times faster than random guessing. For a large fraction of PINs consisting of 4 to 6 digits, we are able to infer them in under 20 and 80 guesses, respectively.

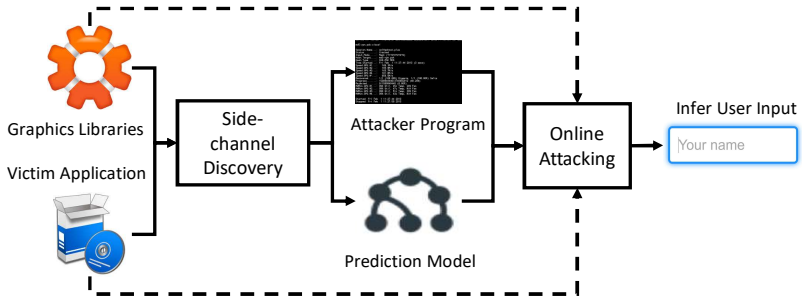


Figure 4.1: Attack workflow.

4.2 Attack Overview

In this section, we provide an overview of the proposed attack starting with the threat model and a high-level description of the underlying intuition. We then briefly describe the individual components of the attack.

4.2.1 Threat Model

The attacker's goal is to passively eavesdrop on the victim's CPU cache looking for sensitive information (e.g., PINs or passwords) the victim entered in an application, which we refer to as the victim application. The attacker is assumed to be local, i.e., the attack process is co-located with a victim process on the same physical device and operating system, (e.g., a piece of malware is installed on the victim's system). We assume that the physical device is a multi-core system, and the attack and victim process can run simultaneously on different cores. The attack process can create a few threads that run continuously in the background. No privileges or special permissions are required.

The attacker should also be aware of when a target victim application is launched and when it is in a sensitive state (e.g., login screen) so as to start the side-channel attack.

Typically the login screens are shown automatically when the app is first launched. However, even if it is not, we can still infer their presence through attacks similar to prior work [36].

Finally, we assume that an attacker has access to the same version of shared graphics libraries used in desktop and mobile applications (these libraries come with the operating system). The attacker has the knowledge of the victim's CPU specification and has access to a device with the same CPU and operating system. These allow the attacker to fully simulate victim's environment offline.

4.2.2 Intuition

By studying how graphics libraries work in general, we observe that when a part of the GUI of an application is updated, only the updated part will be rendered. For example, on-screen keyboard applications will often highlight the key being pressed. In this case, only the highlighted key will be rendered on the screen while other parts of the GUI remain the same. Another example is when a user types a character into an input box, the application will only render the typed character. This precise rendering is necessary for input inference.

Our second observation is that when performing text rendering, the graphics library often renders only the pixels representing the text while ignoring the background pixels. Because of this, the rendering of characters with fewer pixels such as "1" and "i", is considerably faster than rendering more complex characters such as "8" and "w".

Given these observations, if an attacker can measure the time it takes for a victim application to render its GUI, she could potentially use this as a side-channel to infer the user's input to the application. In practice, we envision that the attacker conducts offline profiling experiments to map different user inputs to execution times of subroutines related

to rendering in shared graphics libraries. Later, she performs online attacks by leveraging the prepared mapping to associate measured execution times back to the user's input.

Without any privilege, an attack process cannot directly measure the program state of the victim. Fortunately, a flush+reload cache side-channel attack can be utilized to indirectly measure graphics rendering time through the shared graphics libraries. Note that previously studied flush+reload attacks [105, 51, 80, 107, 72, 79] were successful at checking only the presence and absence of data in the cache. In our attack, we take a step further to measure the execution time of subroutines.

Measuring the execution time of a shared library subroutine requires an attacker to locate at least a pair of instructions (and their corresponding cache lines). However, this can be challenging and tedious. First of all, graphics libraries are complex and graphics rendering often involves multiple libraries. For example, a typical desktop application on Ubuntu Linux will involve libraries including `libgdk-3.so`, `libcairo.so` and `libpixman.so` which have millions of instructions. Manually going through them is not scalable (especially considering that there are many platforms and library versions). Secondly, even if the attacker finds a good target pair to monitor, it is unclear whether it is reliable and effective in practice due to features such as the cache prefetcher. As a result, we need automated discovery and evaluation of good target cache lines to monitor.

4.2.3 Attack Workflow

In this work, we overcome the above challenges and show 1) how we automate the discovery and selection of viable instructions in graphics libraries and 2) how we automatically generate the working exploits of the discovered side-channels.

Our attack is divided into two phases: side-channel discovery (offline) and online attack. Figure 4.1 shows the general workflow of our proposed methodology. During the side-channel discovery phase the attacker selects a victim application and one (or more) shared graphics libraries to target. The goal of this phase is to (i) analyze the victim application's execution, (ii) discover execution-time side-channels inside the graphics libraries and, (iii) craft an attack program (malware) that exploits the discovered side-channel iv) generate a prediction model based on the side-channel for future attacks. Detailed descriptions of the side-channel discovery phase will be described in Section 4.3.

To carry out the online attack on user's device, the attacker runs the malware program alongside the victim application, and performs the flush+reload cache side-channel attack on the vulnerable cache lines. The attacker then uses the prediction model to compute the most likely input from the victim. Finally, the attacker could combine the result with other information (such as a password dictionary) to improve the accuracy. We discuss the details of the design and implementation of the online attack in Section 4.4 and 4.5.

4.3 Side-Channel Discovery

4.3.1 Overview

As discussed in Section 4.2.3, in the profiling phase, our goal is to find a pair of cache lines (x, y) (derived from instruction addresses) in the shared graphic libraries as targets for our flush+reload attack. Here we use the term "cache line" as a memory range in the shared library that occupies the same cache line block in the CPU cache (e.g.,

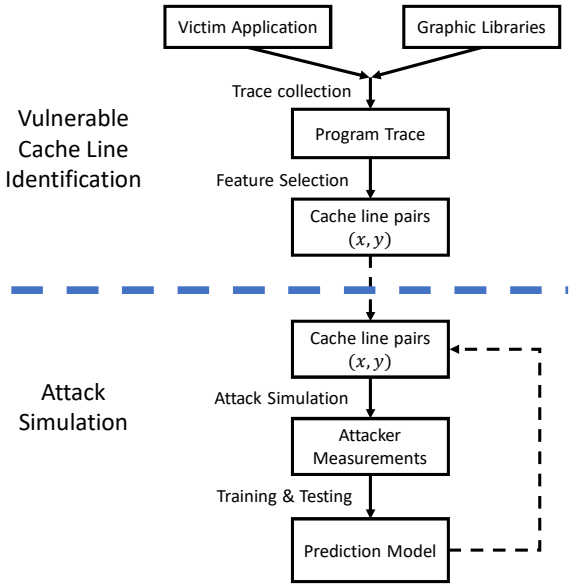


Figure 4.2: Flow chart of side-channel discovery.

0x7f00-0x7f3f). In order to make the attack effective, we need to find the vulnerable cache lines (x, y) such that the time difference between the first access of x to the first access of y (denoted as d_{xy}) "is dependent" on user input. If the attacker is able to reliably measure d_{xy} when users are inputting their PINs or passwords using a cache side-channel attack, they can map the measurements to the original input.

To increase the attack reliability, we need to reduce measurement noise. Measurement noise can originate from two different sources. First, the victim application's behavior combined with modern CPU cache features, (e.g. prefetcher) could significantly decrease the effectiveness of a flush+reload attack, which we refer to as **application noise**. Second, the background processes sharing the same libraries could also introduce noise to the flush+reload attack, which we refer as **system noise**. In side-channel discovery phase, an attacker tries to identify the cache lines least affected by these noises.

Figure 4.2 captures the process we follow to find the vulnerable cache-line pairs that are suitable for attack. This process is designed to minimize the negative impact of both application noise and system noise in a real online attack scenario. Below, we describe the sequence of steps needed.

1. Vulnerable Cache Line Identification

1. The attacker runs the victim application multiple times with different user inputs and collects the program traces for graphics libraries used by the application.

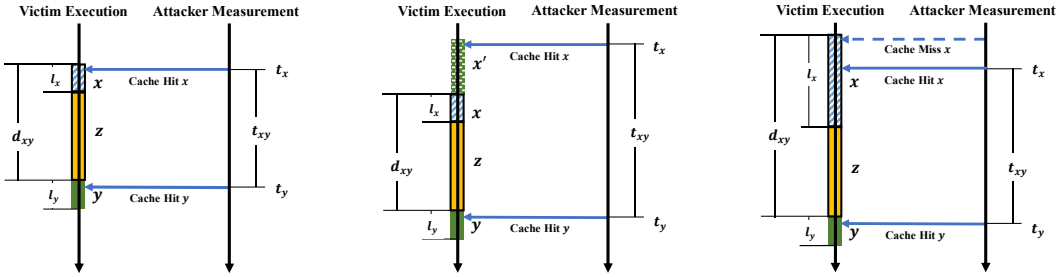
2. The attacker uses a feature extraction algorithm to identify potentially vulnerable cache lines from libraries that are least affected by the application noise. See Section 4.3.2 for more details.

2. Attack Simulation

1. For each pair of cache lines (x, y) identified, the attacker runs a simulated offline attack and collects the measurement times.

2. The attacker builds a key-press prediction model using the collected measurement times (see Section 4.3.3).

3. If the performance of the prediction model is not better than a random guess (e.g., $< 20\%$ for numeric characters 0-9), the attacker selects another pair $\{x, y\}$ and starts over until all the selected cache line pairs are tested. The attacker then picks the cache line pairs with the best performance results. This assures that the chosen cache line pairs are least affected by system noise. See Section 4.3.3 for more details.



(a) Successful attack scenario.

(b) Prefetcher noise due to cache line x' prefetching x. Attacker falsely thinks x is being executed by the victim.

(c) Measurement noise due to l_x too large while flush+reload miss a signal. Attacker's measurement t_x is more likely to be inaccurate.

Figure 4.3: Attack scenario illustration. Victim executes x, z, y sequentially while execution time of z varies depend on user input.

4.3.2 Vulnerable Cache Line Identification

We instrument the victim program and collect the program execution traces with regard to its "cache line accesses" under different user inputs. The cache line trace is a representation of a sequence of instructions being accessed and loaded into CPU instruction by the victim application. For example, on a machine with 64-byte cache lines, if a program executes instructions sequentially from addresses from 0x8020 to 0x80b0, the CPU will load cache lines 0x8000-0x803f, 0x8040-0x807f and 0x8080-0x80bf into the instruction cache. Given a cache line trace, we can compute d_{xy} for all combinations of cache lines x and y.

Next, we design an algorithm to find a pair of cache lines (x, y) such that the distance between the appearances of x and y (denoted as d_{xy}) in the trace varies determin-

Algorithm 1 Feature Extraction and Selection

Input $\mathbb{T}$: Set of traces that are related to user input event.

Output $X = [(IG_{xy}, G_x, G_y)]$: Pairs of cache lines for flush+reload attack, ordered by information gain

```
Result ← ∅
𝔾 ← groupGenerate(𝕋)
for all G_x, G_y ∈ 𝔾 do
    Compute d_xy, l_x, l_y
    if d_xy > d_threshold & l_x < l_threshold & l_y < l_threshold then
        if standard_deviation({d_xy}) ≥ std_threshold then
            IG_xy ← Compute information gain of (G_x, G_y)
            Add (IG_xy, G_x, G_y)) to Result
        end if
    end if
end for
```

istically based on the user's input, thereby providing a potential side-channel. A graphics rendering operation might contain multiple side-channels, meaning that we might be able to find multiple pairs of (x, y). However, not all of these pairs can be used in a practical flush+reload attack to collect accurate measurements because of the following types of **application noise**.

1. **Prefetcher Noise (Figure 4.3(b)).** The cache lines x and y might be prefetched by the prefetcher, when other memory blocks with slightly lower addresses are being fetched. As a result, we might get false hits with the flush+reload attack.

2. **Measurement Noise (Figure 4.3(c).** x and y may be accessed by the victim application (or prefetched) multiple times during the rendering operation. There are chances that we might miss the first access of x or y (due to a cache line being evicted

by victim or background processes before the attacker can reload it) and capture a later access. This will result in an inaccurate measurement with regards to the time interval between x and y.

3. **Prediction Noise.** The flush+reload attack has a limited resolution. Therefore, d_{xy} must be large enough to be captured by flush+reload attack. For good prediction, d_{xy} should significantly differ across different user inputs while being consistent with the same input.

Figure 4.3(a) depicts an example of a victim application's execution trace and one possible attack scenario. In this scenario, the victim application executes code fragments in cache lines x, z, and y sequentially. The execution time of z varies depending on user input, thus creating a side-channel. The execution time of x and y are constant. Therefore, attacker can monitor the time where x and y first appears in CPU cache (flush+reload cache hit) as t_x, t_y respectively and measure $t_{xy} = t_y - t_x$ as a measurement of d_{xy} to infer user input.

However, as depicted in Figure 4.3(b), if the victim accesses cache line x' before x and accessing x' prefetches x into the CPU cache, the attacker might get a cache hit on x while in fact only the code in x' are being executed. This creates prefetcher noise that makes t_{xy} unable to measure d_{xy} accurately.

Moreover, the "flush" operation of "flush+reload" attack takes some time to complete. So there are chances (although rare) that the victim or some other background process may evict the cache line x or y before the attacker can "reload" the cache line and capture the signal. As a result, the attacker could possibly miss x where it is first loaded

into the CPU cache and successfully capture it later on, as depicted in Figure 4.3(c). This creates a measurement noise whose level is determined by the "lifespan" of x or y (denote as l_x, l_y). If l_x and l_y are small, the attacker might miss the signals x or y completely but when it does capture a signal, the signal is guaranteed to be more accurate. On the other hand, if l_x and l_y are large, the attacker has a better chance at capturing the signals and measuring t_x and t_y, and yet the measured result could be very inaccurate. We find that inaccurate measurements are more detrimental to the attacker than missing measurements as the attacker might unknowingly use them for both training and in the actual attack. On the other hand, missing measurements can be compensated by methods such as having the attack process monitor multiple pairs of cache lines, as we will discuss in Section 4.4 and Section 4.5. As a result, our feature extraction algorithm favors a relatively small l_x and l_y.

In order to find a good pair of cache line (x, y) that achieves a good accuracy given the above constraints, we design and implement an algorithm to extract potential targets from program traces. Algorithm 1 captures this logic which we discuss in the subsequent paragraphs.

The first step in the algorithm is to find these cache line accesses in the shared library that are least affected by the CPU cache prefetcher (i.e. prefetcher noise). For this purpose, we put the cache lines into a set of no-conflict groups $\mathbb{G}$. We ensure that accessing a cache line from one group will have no prefetching effect on a different group. For example, according to Intel's optimization manual [6], the prefetcher can prefetch up to 20 consecutive cache lines. Therefore, we ensure that there is at least $20 * cache\_line\_size$ byte gap between cache lines of different groups.

With all groups $\mathbb{G}$ determined, we then compute d_{xy}, l_x and l_y for all pairs of groups G_x and G_y. We only select pairs of groups (G_x, G_y) where the lifespans of both groups l_x and l_y are below a threshold $l_{threshold}$. This ensures a minimum measurement noise level as discussed earlier.

To reduce the prediction noise, we filter out group pairs where d_{xy} is no greater than threshold $d_{threshold}$, as these are too small to be measurable given the limited resolution of the flush+reload attack. We also need to ensure that the d_{xy} is sufficiently different for different inputs. For this, we first use a coarse-grained filter by checking the standard deviation of d_{xy} for different inputs. If it is less than a threshold, we exclude the associated cache line pair.

To further reduce the prediction noise and select the best group pairs suitable for the flush+reload attack, we use "information gain" as a metric for selection [55]. The information gain captures the discriminatory value of a cache line pair by quantifying "how much information the cache line pair gives with respect to uniquely separating the users' inputs". In particular, it is a computation of the reduction in entropy. The cache line pairs that perfectly partition the user inputs will have the highest information gain. Our approach ranks the cache line pairs based on their information gain and selects the top-ranked cache line pairs; those that do not add much information will have a lower score and are removed.

Finally, the attacker will need to perform the flush+reload attack on two specific cache lines viz., $\{x, y\}$ instead of two groups $\{G_x, G_y\}$. Therefore, we pick x and y to be the first cache lines that appear in the trace, and belong to their respective groups.

4.3.3 Attack Simulation

We now have a list of features (cache line pairs) that are derived from the program trace. However, during the real attack, there can still be unpredictable **system noise**. Hence, the vulnerable cache lines identified in Section 4.3.2 might not perform well during the real attack. Therefore, as mentioned earlier in Section 4.3.1, we need to run a simulated offline attack to find the cache line pairs least affected by the **system noise**.

To perform the attack simulation on a cache line pair (x, y) we create an attack process performing the flush+reload attack on one of the CPU cores while running the victim application on another core. Initially, the attack process continuously monitors cache line x and awaits the victim's activity. If the attack process observes a cache hit on cache line x, it records the time of the observation as t_x and immediately switches to monitoring cache line y. If the attacker successfully observes a cache hit on cache line y within a timeout threshold, it records the time of the observation as t_y. If both t_x and t_y are measured, it computes $t_{xy} = t_y - t_x$. Detailed implementations of flush+reload will be described in Section 4.4.2 and Section 4.5.2.

Model Construction. We use the collected measurements to build our key prediction model. We choose the *Random Forest* [32] algorithm for our key prediction model as these classifiers are robust to outliers, and resilient to irrelevant features [32]. We use sklearn [85] implementation of Random Forest with 100 estimators. The performance of the key prediction model was evaluated with 10-fold cross validation [67]. We compute the true positive rate TPR and the false positive rate FPR. TPR refers to the ratio of the total number of correctly identified instances to the total number of instances present in the

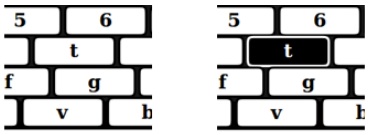


Figure 4.4: Partial graphical user interface of Onboard keyboard and the highlighting effect during key-press.

classification model; the FPR refers to the ratio of the total number of negative instances incorrectly classified as a positive instance to the total number of actual negative instances. A model with a high TPR and a low FPR is considered good for classification tasks. We measure the performance metrics (of the machine learning model) using different cache-lines pairs as features and select the cache-line pairs which result in the highest TPR and a low FPR for the flush+reload attack.

In Section 4.4 and Section 4.5 we describe the detailed implementations on two different platforms (Linux and Android) and demonstrate their effectiveness with real-world attacks.

4.4 Attack I: Ubuntu On-screen Keyboard

In this section, we demonstrate our attack on the default on-screen keyboard (used in Ubuntu) to extract a user's password. Such virtual keyboards are necessary in a touch screen scenario. Even without the touch screen, it is recommended that we use such virtual keyboards as a more secure alternative for entering private credentials [15] because it is less prone to various attacks such as keyloggers (e.g., [92, 112]). We evaluate our attacks on a desktop machine with 3.40 GHz Intel Core i7-4770 CPU, which has a 8MB L3 cache

Table 4.1: Graphic library versions used in Onboard attack.

Library	Version	Description
libgtk-3	0.1800.9	Multi-platform toolkit for creating GUI.
libgdk-3	0.1800.9	A wrapper around the low-level functions provided by the underlying windowing and graphics systems.
libcairo	2.11400.6	Provides primitives for two-dimensional drawing.
libpixman-1	0.33.6	A low-level software library for pixel manipulation.
libfreetype	6.16.0	Render text onto bitmaps, and provides support for other font-related operation

and 64 byte cache line size. The GPU on the machine is Nvidia GeForce GT 635 and the main display has a resolution of 1920x1080 pixels. Table 4.1 lists the versions of several graphic-related libraries. We study the "Onboard" input method editor (IME) that comes with Ubuntu Desktop 16.04. Figure 4.4 shows the graphical user interface of Onboard.

When a key is pressed by the user, a "highlight effect" of the pressed key will be rendered on the keyboard. The highlight effect includes (i) a color change of the border, (ii) a fill of the key (iii) a color change with regards to the character represented by the key. Since different characters are composed of different numbers of pixels, we suspect that there are side-channels within the highlight rendering process.

4.4.1 Side-Channel Discovery

For our offline trace collection, we use Intel PIN [13] to perform binary instrumentation of the graphics libraries listed in Table 4.1 (we identified them by checking the description of the every library loaded in the Onboard keyboard process). The instrumentation allows us to collect full-instruction traces of the keyboard process during a user's key

Table 4.2: Top cache line pairs selected for Onboard IME attack

#	Cache Line	Library	Function Name
1	0x75a40	libcairo.so	_cairo_surface_create_scratch
	0x69e40	libcairo.so	_cairo_scaled_font_map_lock
2	0x69e40	libcairo.so	_cairo_scaled_font_map_lock
	0x41f40	libcairo.so	_cairo_intern_string
3	0x24440	libcairo.so	_cairo_clip_copy_with_translation
	0xbe000	libcairo.so	_cairo_ft_unscaled_font_lock_face
4	0x6b900	libcairo.so	_cairo_path_fixed_approximate_stroke_extents
	0x41700	libcairo.so	_intern_string_pluc
5	0x6a5c0	libcairo.so	_cairo_scaled_font_thaw_cache
	0x41700	libcairo.so	_intern_string_pluc

press. We then convert the instruction trace to the corresponding cache line trace of the victim application. Such a trace is devoid of any form of cache pollution. (e.g., from PIN, background process, prefetcher, etc.). From the collected trace we find a very large number of cache lines being accessed by multiple graphics libraries. In `libcairo.so` alone we find 2591 cache lines, which can create around 6.7 million cache line pairs.

We then run our feature selection algorithm on the collected trace and attempt to discover side-channels. Using the procedure discussed in Section 4.3.1, we first put all cache lines into groups to eliminate prefetcher noise. In the case of `libcairo.so`, the cache lines form a total of 150 groups, which can create 22350 cache line pairs. The algorithm then filters pairs of cache lines that are not suitable for our attack. The algorithm left us 1488 cache line pairs, ranked by information gain. Interestingly, we find that the majority of the identified cache lines are located at the beginning of functions. This is reasonable since the instructions of functions are organized contiguously in the memory address space.

Cache lines located at the beginning of functions are much less likely to be affected by the prefetcher and thus have a better chance of being selected by our algorithm.

Interestingly, we find that out of the few graphics libraries only `libcairo.so` produces good cache line pairs. Upon closer inspection, it turns out that the measured execution time between cache lines pairs from GDK and GTK libraries are not consistent for the same key press. Cache lines corresponding to libfreetype.so are only accessed when a key is first pressed during the lifespan of the Onboard process (we'll discuss exploitation of Freetype library in Section 4.5). Cache lines corresponding to libpixman.so are seldom accessed under the default Linux X server graphics architecture and only play a role under the Wayland architecture (an alternative to X server) as will be discussed in Section 4.7).

From the cache lines identified in `libcairo.so`, we further perform an offline simulation attack on the top 100 cache line pairs to filter out cache-line pairs that do not produce good results. For each lower-case letter and number, we collect 50 measurements by having one attacker thread running in the background monitoring the selected cache line pairs (as discussed in Section 4.3.3). Next, we build our prediction model with Random Forest classifier on these measurements and evaluate it with a 10-fold cross validation [67]. We select the cache-line pairs and the prediction model with the highest true positive rate. Table 4.2 shows the top 5 cache-line pairs that performs best during attack simulation.

To understand the underlying cause of the input-dependent execution time, we choose to inspect the source code of the corresponding address pairs. According to Table 4.2, cache line pair #1 and #2 clearly corresponds to two separate side-channels, one from 0x75a40 to 0x69e40 and another from 0x69e40 to 0x41f40. We find these two side-

channels are both part of function `cairo_show_glyphs()`. This function is tasked with computing the rendering result of a given character (glyph) and send the rendering command to Linux X server. The function will first load the pre-computed font data (in which the input character will be rendered) and compute a scaled "pattern" matrix via a series of matrix transformations and multiplications, amounting to the first discovered input-dependent execution time (between 0x75a40 and 0x69e40). This operation involves over 100,000 instructions and its complexity depends on both the font used and the character to be rendered. Next, before contacting X server to render text on the screen, `cairo_show_glyphs()` will first render the computed "pattern" matrix on its own `cairo_surface_t` struct. This operation takes around 10,000 instructions to complete and its complexity also depends on the input character, thus creating the second side-channel (between 0x69e40 to 0x41f40). Pairs #4 and #5 also cover the second side-channel while pair #3 covers both side-channels.

In theory, one cache-line pair is sufficient. In practice though, due to measurement noises (and the possibility of missing signals during flush+reload), we simultaneously monitor two cache-line pairs for redundancy. We further use them to train a new machine learning model for the actual attack. Empirically, we observe that the addition of more features (more cache line pairs) improves the prediction accuracy. However, monitoring too many cache line pairs results in much noisier measurements (as flush+reload itself as well as context switches create noises). Ideally, for the best attack resolution, an attacker thread should only monitor one cache line pair on a CPU core. On our desktop machine, we find that the measurement becomes noisy when we run the flush+reload attack to monitor more than two cache-line pairs. We select the top cache line pair #1 and #2 from Table 4.2 for

our side-channel attack. Luckily, we also find that when a user presses a key on Onboard, there will be two rendering operations. The first operation will render the highlighted key while the second operation will reverse the highlight effect. Both operations will generate a signal measurable using the selected cache line pairs. Therefore, the attacker can capture 4 measurements for a single key press.

4.4.2 Flush+reload Evaluation

We first evaluate the flush+reload attack in a controlled environment. We create a controlled victim process that accesses a cache line x and collects the timestamp of the access as the ground truth. Meanwhile, the attack process tries to capture the time when x is accessed by the victim. To achieve this, the attack process "loads" a memory address in the cache line x, checking whether a cache hit or miss has occurred. Then, the attacker thread executes the CLFLUSH instruction on the address just read. We find that we need to introduce a short delay after the CLFLUSH instruction and before the next load operation in order to capture the signal reliably. Similar to previous works [51], we use `sched_yield()` to introduce this delay.

The more delay we introduce, the better the chance that the flush+reload attack will be able to capture the signal. However, a larger delay also means a longer time for each round of flush+reload; this reduces our measurement resolution. Figure 4.5 demonstrates the effect of `sched_yield()`. In our attack, we set the number of `sched_yield()` calls to 3, which is the setup that can reliably capture the signal and maintain a reasonable measurement resolution at the same time.

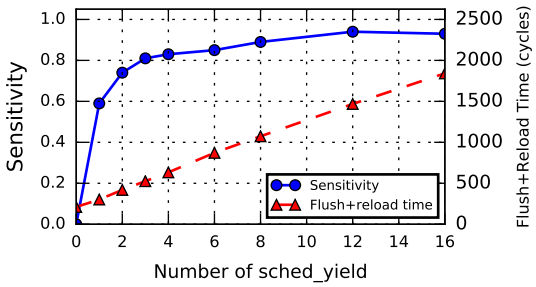


Figure 4.5: Effect of *sched_yield*()

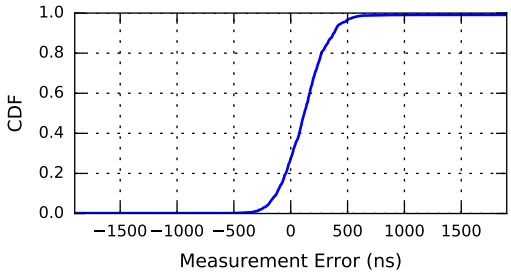


Figure 4.6: CDF distribution of measurement error.

Figure 4.6 presents the cumulative distribution function (CDF) of the measurement error. We see that in 90% of the experiments, the attacker is able to measure the execution time of the target operation with an absolute error of less than 700 ns. Given that a single round of flush+reload takes around 500 ns, we conclude that our cross-process execution time measurement is accurate.

4.4.3 Password Inference Attack

We then run our attack to infer a user password inputted using the Onboard keyboard. We have two attacker threads each monitoring the selected cache line pairs (in Table 4.2) respectively, while a user is inputting passwords. After the attacker thread col-

lects the measurement data during the user's input, we use the trained prediction model generated via the offline simulation to predict the user passwords based on the measurements.

For demonstration, we only consider lower-case letters and numbers for the password field inputted using the Onboard keyboard. We choose the list of most common passwords as our dictionary [7]. This dataset contains 10,000 unique passwords, with 9984 of them composed of lower-case letters and numbers. Note that for all our attacks except for the one augmented with dictionary, we assume the attacker has no knowledge of the passwords in the dataset.

Single Character Prediction Accuracy

Single Login Attempt. First, we test our password prediction capability when we capture a single login attempt from the user (we might be able to observe multiple attempts over time). For each character, we perform the attack multiple times and test whether the attacker can correctly predict the character within a certain number of guesses. Figure 4.7 demonstrates the single-character prediction accuracy for numeric characters. We observe that we can reach 90% accuracy in 10 guesses for all numeric characters. Some characters (e.g., "2") can be predicted more accurately than characters like "0" and "4"; this can be helpful in inferring PINs. However, predicting lower-case letters are much more difficult. With 10 guesses we can only reach 70% accuracy. Some characters such as "u", "v", and "w" requires more than 30 guesses while characters like "i", "o", and "y" can be predicted pretty accurately in fewer guesses. When it comes to lower-case letters some of them look similar and the measured time difference is often overwhelmed by the noise. As we will show

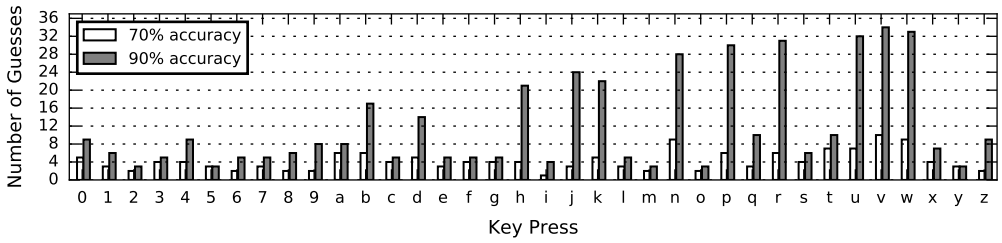


Figure 4.7: Onboard IME: Single login attempt. Number of guesses required to reach 70% and 90% accuracy.

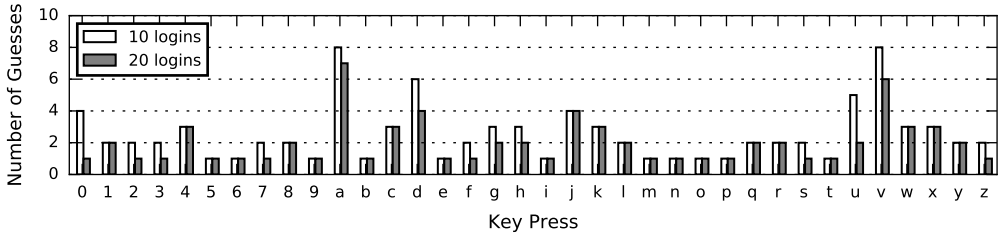


Figure 4.8: Onboard IME: Repeated login attempts. Number of guesses required to reach 100% accuracy.

next, the limited measurement resolution and noise are the main reasons why the accuracy of observing a single login is not as high.

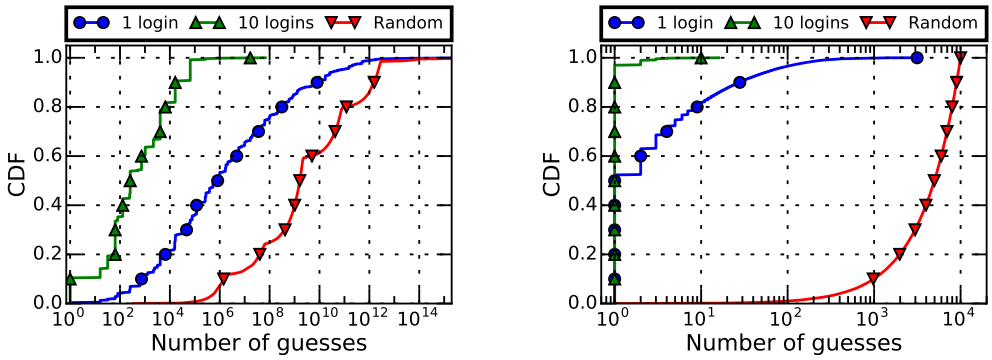
Repeated Login Attempts. A password is often reused or repeatedly inputted by the user. An attacker has the opportunity to obtain measurements of the user's repeated login attempts using the same password. This gives the attack more potency i.e., by combining multiple measurements together the attacker can make better predictions (e.g., because the noise can be corrected). Suppose the user inputs password $p = b_1 b_2 ... b_n$ N times. At the jth instance, suppose our prediction model generates confidence vectors $C_{j,1}, ..., C_{j,2}, C_{j,n}$. We can then combine these guesses by simply adding the prediction confidence values together, letting $C_{comb,k} = \Sigma_{x=1}^{N} C_{x,k}, k = 1, ..., n$ to be the aggregated confidence vector.

We study the per-character prediction accuracy when we have 10 and 20 measurements of the same character, as shown in Figure 4.8. We can see that more measurements leads to higher prediction accuracy. With 10 measurements, all numeric characters and most lower-case letters can be predicted with 100% accuracy within 4 guesses. Characters like "a", "d" and "v" are a little harder to predict than other characters, yet the attacker can still predict them with 100% accuracy within 8 guesses. With 20 measurements, the prediction accuracy is slightly improved over 10 measurements. 17 out of 36 characters can be predicted perfectly in the first guess.

Multi-Character Prediction Accuracy

To guess the entire password correctly, we need every character to be guessed correctly. This means that the total number of guesses is bound by the prediction accuracy of the worst character. Specifically, if the worst character takes k guesses to achieve a 100% accuracy, then the total number of required guesses will be k^n where n is the number of characters in the password. This is because we cannot know a priori which character is the worst during guessing and will have to exhaust all possibilities.

In the best case when the attacker can observe 20 login attempts, the number of needed guesses is then 5^n, which is a drastic improvement from the original 36^n. Figure 4.9(a) further illustrates our results. We observe that having multiple measurements of the same password significantly improved our password-guessing effectiveness. On average we need 1000 times fewer guesses to infer a password compared to the case where we used a single input, and 1,000,000 times faster over a random guess. Finally, 40% of the passwords are guessed within 100 attempts.



(a) Password inference attack without dictionary.

(b) Password inference attack with password dictionary.

Figure 4.9: Onboard IME: The cumulative distribution function for the number of guesses needed to infer passwords.

Augmentation with Dictionary Attack. Password characters are often non-independent events. To better utilize this dependency between password characters, attackers often use dictionary attacks to reduce the search space. Being able to guess the password in fewer attempts is useful as an account may be locked after a few failed login attempts. Our attack can work very well in conjunction with dictionary attacks to boost its effectiveness. Let $p = b_1 b_2 ... b_n$ be the target password and $C_1, ..., C_n$ be the prediction confidence vectors (single-input or combined). Let $conf(a, C)$ be the confidence value of character a in confidence vector C.

For each n-character password $w = w_1 ... w_n$ from the dictionary, we can compute the confidence c_w of w being the correct password, to be $c_w = \Sigma_{i=1}^{n} conf(w_i, C_i)$. We then rank all possible n-character passwords based on the confidence values c_ws, and try them in order. If the correct guess has the kth-highest confidence score, we will need k guesses.

Table 4.3: Example dictionary-assisted password guessing attack for password "hello".

Input	Confidence Vector (Partial)							
	e	h	i	j	l	o	s	y
h	0.0	0.39	0.0	0.23	0.0	0.0	0.0	0.03
e	0.21	0.0	0.0	0.0	0.0	0.0	0.0	0.03
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.0
l	0.0	0.0	0.05	0.0	0.37	0.0	0.07	0.06
o	0.0	0.0	0.0	0.0	0.0	0.15	0.0	0.0

Rank	Dictionary Words	Confidence Value
1	hello	$0.39 + 0.21 + 0.37 + 0.37 + 0.15 = 1.49$
2	jelly	$0.23 + 0.21 + 0.37 + 0.37 + 0.0 = 1.18$
3	hills	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$
4	holly	$0.39 + 0.0 + 0.37 + 0.37 + 0.0 = 1.13$

Table 4.3 shows an example of dictionary attack. When user inputs password "hello", the attacker would be able to obtain five different measurements each corresponding to one character in the password. The attacker will then use the prediction model to generate five confidence vectors $C_1, ..., C_5$, where each character in the password alphabet is given a confidence value. Next, the attacker look up all the passwords in the dictionary with a length of 5 and compute its confidence value. The attacker finally ranks all the 5-character passwords based on their confidence values and proceeds to use them as guesses in order. In this example, "hello" has the highest confidence compared to the other 5-length passwords in the dictionary. As a result, the attacker can guess "hello" in the first attempt. Our

algorithm ranks the guesses based on confidence relating to each character. Note that we use the sum of individual confidences instead of product as we do not want to penalize a guess for one poorly predicted character. For example, in Table 4.3, suppose the correct password is "jelly". Multiplying the confidences will result in a low rank of the word "jelly" for merely a poorly predicted "y".

We compare our password guessing approach with random brute-force guessing with dictionary. With the random approach, the attacker needs to look up all the passwords in the dictionary and guess them in random order. Therefore, the number of guesses required with the random approach is a random number between 1 and the total number of passwords in the dictionary (10,000 in our case).

Figure 4.9(b) demonstrates the result. With one input of the password, our approach can guess 50% of the passwords correctly in the first attempt and 80% the passwords within 10 guesses. On the other hand random guesses can hardly crack anything within 10 guesses. If attacker can measure 10 login attempts, 95% of the passwords can be cracked in the first guesses. We do acknowledge that this result is dependent on the size of dictionary we are considering.

4.5 Attack II: Android Application

In this section, we demonstrate our attack on two Android applications to extract a user's password and PIN, respectively. For this demonstration, we use a Nexus 6P running Android 8.0, as the victim's device. The victim apps are the CapitalOne banking and the Reliance Global Call [8] (an app similar to the Skype) app. With both apps, every time the

Table 4.4: Cache line pairs selected for CapitalOne attack

#	Cache Line	Library	Function Name
1	0x176a80	libskia.so	SkScalerContext_FreeType_Base::generateGlyphImage
	0xef440	libskia.so	SkMask::getAddr
2	0x109e40	libskia.so	SkGlyph::computeImageSize
	0xca8c0	libskia.so	SkAAClipBlitter:: SkAAClipBlitter

app opens, it will require a user to input his/her username and password/pin. For security reasons, the exact password and pin are not normally rendered on the screen and often replaced with stars or dots. However, to prevent input errors, Android, by default, makes any input character visible for one second before masking it.

4.5.1 Side-Channel Discovery

For offline trace collection, we instrument the graphics libraries (e.g. `libskia.so`, `libfreetype.so`, etc.) from Android AOSP source code, recompiled these libraries and loaded them on to the victim device. The instrumentation targets various functions and records the timestamp each time an instrumented function is called. The instrumentation will introduce a small overhead but it does not thwart the feature selection algorithm.

We run our feature selection algorithm on the collected traces and identify top function pairs suitable for attack. We then translate the function names to cache line addresses and perform the offline simulation attack to filter out cache-line pairs that do not produce good results. Normally, the shared library is stripped and thus, not all function names to cache line translations will be straightforward. However, we find that graphics libraries such as `libskia.so` contain enough symbols for us to translate most of the selected

functions. Similar to Onboard attack, we then select the top two cache-line pairs for our attack. Table 4.4 shows the two cache-line pairs and the corresponding functions.

Upon analyzing the function calls made between the identified cache lines, as well as the source code of related graphics libraries, we find that the cache lines actually measure an exploitable side-channel in the *font translation* process from the standard Android Freetype library. Font translation is long in duration and happens whenever an application attempts to render any text (e.g., characters) for the first time. It works by translating the character representation into the graphical representation (glyphs). In Freetype library this is accomplished via `FT_Outline_Decompose()`. Depending on the font and the input character, `FT_Outline_Decompose()` will invoke a series of functions (such as `gray_set_cell()`, `gray_hline()`) to compute the translation result [5]. Each Android process keeps the translation result in its memory space. Thus, the next time the same character with the same font type and size is to be rendered, there is no need to execute font translation again. As a result, the attacker can only capture the first appearance of each character using the this side-channel. However, many sensitive applications, CapitalOne and Reliance Global Call included, require the user's login information right upon the apps' start. At this point, the font translation has not been performed for most characters (especially the font type and size in the password box), leaving a window for attacker to extract sensitive information via timing measurements of font translation.

It is worth mentioning that the software keyboard app similar to those on Linux, is also an excellent target for this attack. However, the software keyboard is maintained in a dedicated long-running process and the font translation results are cached throughout the

lifetime of the process. This means that most of the characters should have been translated by the time the attack is launched.

Note that there are also side-channels discovered on Android that get exercised every time when a key is pressed (regardless of whether it is a pressed for the first time). Unfortunately, the limited measurement resolution on ARM makes them un-exploitable.

4.5.2 Evict+reload Implementation

The ARM architecture poses several technical challenges. First of all, the ARM architecture does not include a CLFLUSH instruction. As a result, we cannot perform a flush+reload attack. Fortunately, we can still perform the evict+reload attack by creating a set of memory blocks (eviction set) that can evict the target cache line. Our implementation is similar to Gruss et.al.'s implementation [72].

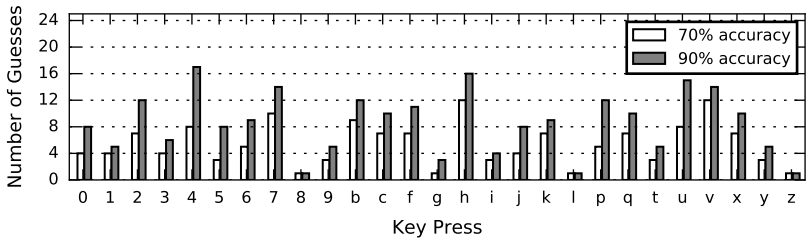
Eviction is slower than the CLFLUSH instruction. On the LG Nexus 5X, each round of evict+reload takes 10μs to 13μs, while on our x86 desktop machine each round of flush+reload takes only around 0.5μs. As a result, the attack resolution of evict+reload is lower than that of flush+reload. This means that our attack will be less effective for faster graphic operations such as text rendering. The font translation process takes a long time to compute and thus, the evict+reload attack is able to exploit its associated side-channel.

The evict+reload attack is easier to implement when the attack process can read its page table and know the physical address of each memory block in the eviction set. However, later Android versions no longer grant user process read access to its page table. Still, Oren et.al., [79] demonstrated that evict+reload can still be performed without an attacker knowing the virtual-to-physical address mapping.

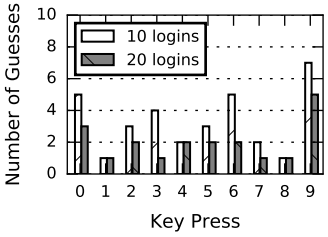
Another challenge in realizing the evict+reload attack on ARM is that many ARM CPUs do not have an instruction-inclusive shared L2 cache. In addition, Green et.al. [46] have shown that there are other features in the ARM CPU implementation that make an attack more difficult. Fortunately, most ARM CPUs are cache coherent. When a process accesses a cache line not currently cached in its own core, the CPU will try to fetch it from other cores in case other processes are accessing it. If successful, the resulting access will be much faster than access from memory. This feature has been exploited in recent works [107, 72] and we also rely on it.

However, the phones we tested (Nexus 5X and 6P) have an additional challenge in that the attacker and victim cannot have a shared L2 cache. Nexus 5X adopts big.LITTLE technology [10]. It has 2 Coretex-A57 "big" cores sharing a single unified L2 cache and 4 Coretex-A53 "LITTLE" cores sharing 2 unified L2 caches. It appears that only the cores among big and LITTLE have the proper cache coherency protocol that can be exploited, which requires the attacker and victim to be on different classes of cores. Further, we find it interesting that the eviction takes significantly more time on the big cores (likely due to their cache replacement policy). As a result, we pin the attack threads on the "LITTLE" cores while leaving the victim on the "big" cores.

The net effect of this setup is that we are unable to evict the victim's L1 and L2 cache. As a result, when checking for the victim accesses relating to the cache line targeted by the evict+reload attack, the attacker will keep getting cache hits for a short while even when the victim is not accessing it anymore. Fortunately, our attack only focuses on capturing the target cache line's first appearance. As a result, the attack is not affected



(a) Number of guesses needed to infer an input character correctly.



(b) Number of guesses needed to infer a digit.

Figure 4.10: Android: Number of guesses need to infer each character correctly.

by the lack of victim-core eviction. Since L1 and L2 have limited sizes, shortly after the victim finishes executing the target function, the cache will be evicted by other functions of the victim automatically.

4.5.3 Password and Pin Inference

Password Inference. We attack a user who is launching a new CapitalOne application process and inputting a password from the common password list [7] (the same 10,000 password dataset as used in Section 4.4). During this process we have the attack threads running in the background and collecting measurements. Next, we use the trained prediction model to predict the user's key press based on the measurements collected.

In this attack, we measure the font translation time when rendering a character on the screen. This process only happens when the character is inputted for the first time (i.e., even if a character is inputted multiple times, the attacker can only perform its timing measurements when it is first inputted). In addition, the CapitalOne app renders "Username" and "Password" in respective input boxes on startup. Therefore, the attacker will not be able to measure the font translation time for characters "U", "P", "a", "d", "e", "m", "n", "o", "r", "s" and "w" (as they share the same font type and size with the actual password). Interestingly, the attack can still be performed with these restrictions. The attacker can easily use evict+reload attack to monitor when a key-press is happening and combine the result with the font translation timing measurement. If the attacker observes a key press but does not get a timing measurement, the attacker can infer that the inputted character is either in the list of pre-rendered characters or something that the user had previously inputted.

Table 4.5 shows an example attack scenario when "hello" is inputted:

- User inputs "h". "h" is not pre-rendered by the CapitalOne app and the attacker will be able to get a measurement. Suppose our prediction model guesses it as "0".
- User inputs "e", which is pre-rendered. The attacker will detect a key press but will have no measurement of the font translation time. The attacker will simply guess it as one of the rendered characters (including the pre-rendered characters and the inference made with regards to the first character i.e., "0"). Suppose we guess "e".
- User inputs "l", which is not pre-rendered. The attacker will be able to get a measurement. Suppose we guess it as "l".

Table 4.5: Example password guessing attack for password "hello".

Input	**Potential Guesses**	Guess	**Description**
h	0,p,h	0	Predicted by model.
e	a,d,e,m,n,o,r,s,w,0	e	Pre-rendered or same as 1st character.
l	l,1,7	l	Predicted by model
l	a,d,e,m,n,o,r,s,w,0,l	0	Pre-rendered or same as 1st/3rd character.
o	0,o	o	Cannot be "0" since its guessed as 1st character.

- User inputs "l", which is already rendered during the previous keypress. The attacker will simply guess it as one of the rendered characters (including all pre-rendered characters and the guesses of the 1st and 3rd characters viz., "0" and "l"). Suppose we guess it as "0".

- User inputs "o", which is not pre-rendered. Attacker will be able to get a measurement and guess it to be either "0" or "o". However, since the belief is that "0" is already rendered, it is inferred that this character can only be "o".

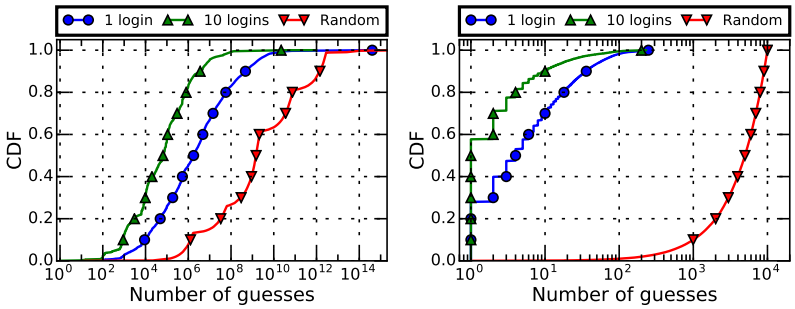
Figure 4.10(a) shows the number of guesses required to predict individual characters. We omit the pre-rendered characters from this figure as the process of guessing them is simply a random selection from a list of known pre-rendered characters. We assume that the attacker can measure 10 login attempts from the user. As we discussed in Section 4.4, having more measurements improves the attacker's accuracy drastically. According to Figure 4.10(a), in general, the result is worse than the Linux Onbard keyboard attack primarily due to the limited measurement resolution of evict+reload. Still, most characters can be predicted with 90% accuracy within 10 guesses. Some characters such as "4", "7", "h",

"u", "v" are harder to guess accurately as they are often confused with other characters. For example, "b" and "q" have a similar shape, thus their font translation times are close to each other. It's very difficult to distinguish them. On the other hand, characters such as "1", "8", "l" and "z" can be predicted very effectively due to their rather unique shape and font translation time.

We compute the number of guesses needed to infer a complete password using our attack model on the CapitalOne application, similar to the previously discussed Ubuntu Onboard keyboard attack Section 4.4.3. We compare our password inference capability with a random brute-force guessing in Figure 4.11(a). We see that using our attack model, the number of guesses needed to infer the password is 10,000 times less than the number of guesses needed with random guessing.

We also compare our password guessing capability in conjunction with a dictionary. As discussed in Section 4.4.3, attackers often use dictionary attacks to reduce the search space. We present our results in Figure 4.11(b). When an attacker is able to capture one login attempt of the password, our approach can infer 30% of the passwords in the first guess and 70% the passwords within 10 guesses. With 10 login attempts captured, our approach can guess 60% of the passwords in the first guess and 90% the passwords within 10 guesses.

PIN Inference. In this section, we exploit the login process of the Reliance Global Call application [14] (a very popular VoIP app), with a similar PIN input box and graphics libraries. Reliance asks users to enter a PIN every time the app opens. We generate random PINs of length 4 and 6 for us to infer.



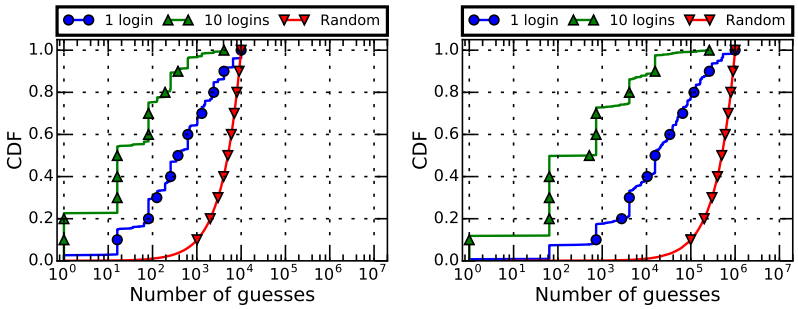
(a) Password inference attack without dictionary.

(b) Password inference attack with password dictionary.

Figure 4.11: Android: The cumulative distribution function for the number of guesses needed to infer passwords.

Similar to the password inference attack, we measure the font translation time when rendering a digit on the screen using the same cache line pairs listed in Table 4.4. We assume that the attacker is able to capture the user inputting the PINs multiple times. These measurements are then fed to the prediction model generated in the offline simulation to predict the users' PINs.

Figure 4.10(b) shows the number of guesses required to guess an individual digit with 100% accuracy. We observe that with 20 user logins, 4 of the digits can be inferred in one attempt and 8 of the digits can be inferred within 2 attempts with 100% accuracy. We next use our prediction model to guess the entire PIN of 4-digit and 6-digit lengths. From Figure 4.12(a) we see about 20% of the 4-digit PINs can be cracked in a single attempt, and 55% of the 4-digit PINs can be cracked in 20 attempts or less. From Figure 4.12(b) we notice that more than 50% of the 6-digit PINs can be inferred in less than 80 attempts. The overall attack success rates in both cases are five to six orders of magnitude better



(a) 4-digit PIN (b) 6-digit PIN

Figure 4.12: Android: The cumulative distribution function for the number of guesses needed to infer the PINs of different lengths.

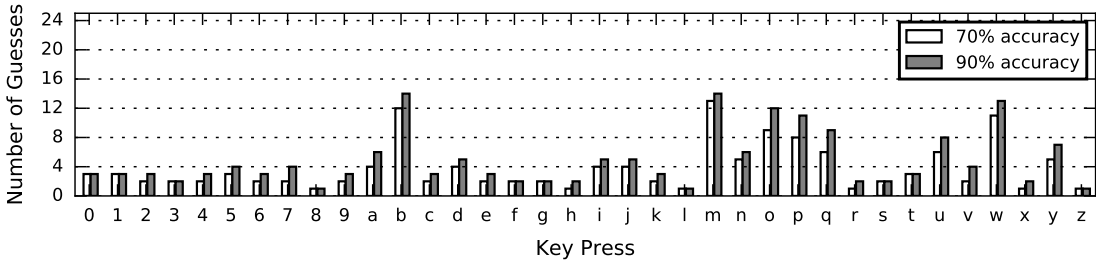


Figure 4.13: CITIC mobile banking: Number of guesses needed to infer an input character correctly.

than the random brute-force attack. Consistent with previous results, we observe that the prediction rate improves as the number of observed login attempts increases.

4.5.4 Attacking Built-in Keyboards

to prevent malicious keyboard applications from recording user's input, some banking apps have built-in keyboards for entering passwords/PINs. Ironically, these keyboards are more vulnerable to our font-translation attack for the following reasons: 1) The banking app is not a long-running processes like a regular keyboard app. This means every time it

launches anew, font translation need to be redone. 2) The keyboard often have a unique font type and size. There is no pre-rendered characters.

We perform our attack on CITIC mobile banking app [2], a popular banking app that has over 8 million downloads. The app uses its built-in keyboard for password entering. Whenever a key is pressed, a pop-up echo containing a enlarged version of the pressed key will be rendered. The password input box, however, will not render inputted character. The rendering procedure is different from CapitalOne and Reliance. Therefore, we perform side-channel discovery as previously discussed and selected the following pairs of functions:

1. `SkScalerContext_FreeType_Base::generateGlyphImage` (`libskia.so`) and `GpuPixelBuffer::map` (`libhwui.so`);
2. `gray_set_cell` (`libft2.so`) and `FontRenderer::cacheBitmap` (`libhwui.so`).

Similar to our CapitalOne attack, we assume that the attacker can measure 10 login attempts from the user. The measurements we obtain from CITIC however, are noisier than the CapitalOne. So we evaluated our key prediction model with Random Forest and Boosting [41] algorithms. Boosting outperforms Random Forest for CITIC app and we selected it to build a key-prediction model. Boosting is a machine learning ensemble algorithm that convert weak learners (high bias, low variance) to strong ones and is resistant to over-fitting [41].

Figure 4.13 shows the character guessing accuracy for our attack on CITIC app. Comparing with Figure 4.10(a), we first find our attack on CITIC can capture all 26 lower-case characters. There's no pre-rendered characters like CapitalOne because the CITIC keyboard uses a unique font. Additionally, We find that the prediction accuracy for CITIC

attack is better than CapitalOne attack, where the majority of the characters can be predicted with 90% accuracy within 4 guesses. This because the rendered echo in CITIC keyboard is much larger than the text echo displayed in CapitalOne password input box, making evict+reload more effective in capturing the stronger signal.

4.6 Related Work

There has been an abundance of existing work on CPU cache side-channel attacks, most of which target encryption keys. For example, Tromer et al. [94] and Osvik et al. [80] demonstrate how the prime+probe attack could be used to break AES by locating memory accesses in the AES lookup table. Zhang et al., [109] [108] show that prime+probe attacks can even cross VM boundaries and perform cross-tenant attacks on PaaS (Platform as a service) clouds. In our work, we use the flush+reload attack, which was first introduced by Yuval et al. [105]. In their work, the authors demonstrate that the flush+reload attack can be used to attack encryption applications such as GnuPG. Gulmezoglu et al., [54] design and showcase an improved flush+reload attack on AES.

A closely related work by Gruss et al. [51] proposes a cache template attack which aims to discover input-dependent cache line accesses automatically in shared libraries. This methodology does not leverage unique characteristics of graphics libraries and therefore misses the opportunity to measure input-dependent execution time. GDK has patched their attack and we're no longer able to find input-dependent cache line accesses. Our attack differs in that we do not focus on unique memory accesses; instead, we rely on the difference in execution times to infer user inputs. Furthermore, we demonstrate that given a conducive

application, we can achieve much better accuracy. On average we reduce the entropy per character for a random password from $log_2(36) = 5.16$ to ($log_2(10) = 3.32$ – $log_23 = 1.58$), while Gruss et al. only reduce the entropy from $log_2(26) = 4.7$ to $log_2(16) = 4$.

We perform our attack on ARM CPUs with an instruction non-inclusive last-level cache. This has been shown to be a minor hurdle for cache side-channel attacks. There have been solutions for both ARM and x86 that primarily leverage the cache coherency protocol among different last-level caches. For example, it was recently shown that the latest non-inclusive last-level cache employed by x86 CPUs can also be attacked [104]. Several researchers demonstrate the possibility of performing cache side-channel attacks on ARM. Zhang et al., [107] design and implement a return-oriented flush+reload attack on ARM, which is essential due to the lack of an data+instruction inclusive last-level cache. Both works utilize the cache coherency policy to monitor victim applications' instruction cache access and we adopted a similar methodology. Gruss et al., [72] perform a systematic study on cache side-channel attacks on the ARM architecture, discussing both the prime+probe and the flush+reload attacks. Our work builds on similar ideas. We had to also deal with the fact that there is no shared L2 cache between the attacker and victim.

In addition to the prime+probe and the flush+reload attacks, researchers also explore other potential side-channel attacks related to the CPU cache. Gruss et al., [50] propose the flush+flush attack, which utilizes the timing side-channel of CLFLUSH instruction under different cache states. Unfortunately, on our machine this attack did not work as reliably as the flush+reload attack. In addition, they also discover a timing side-channel on prefetch instructions [49] and utilize this side-channel to perform address translation

towards breaking ASLR. Lee et al., [68] and Wang et al., [101] show that branch predictors also contain side-channels that can be used to attack secure systems such as SGX.

There are studies on the automated discovery of cache side-channels. For example, Gorka et al., [59] utilize dynamic taint analysis to locate cache side-channels in crypto libraries. Wang et al., [100] model the cache behavior and use symbolic execution in conjunction with their model to discover crypto-related vulnerabilities. These approaches discover only the presence and absence of a unique cache line access to decide if any side-channel is present. Here, we investigate a unique execution-time-based side-channel in shared libraries. Further, we not only automatically discover such side-channels but also generate and evaluate the exploit automatically.

There are other orthogonal research studies exploiting different types of side-channels (e.g., keystroke sounds [26, 112, 56]; electromagnetic waves [97]; vibrations [75] etc.). All these side-channel attacks need physical proximity to the target device. Researchers have also introduced new types of attacks to guess sensitive user input using motion sensors [35, 82] on smartphones or inter-keystroke timings [92]. However, the success of such attacks is dependent on individual users' typing habits. Unlike these attacks, our attack does not need access to a physical device nor is dependent on user behavior. Additionally, defenses mitigating inter-keystroke timing attacks [89] cannot prevent our attack.

4.7 Discussion and Future Work

Measurement challenges: Our attack is sensitive to measurement resolution and noise. For example, we found that with the Linux Wayland architecture, there is a side-channel in `libpixman.so` when it renders text for applications such as Gedit, the Gnome Terminal. We also find that `libskia.so` can perform text rendering pixel-by-pixel for Android applications (in addition to the font translation that is triggered only for the first time a key is rendered within the same process). However, out measurement resolution is too low to perform a reliable attack on these operations. One interesting observation we had is that the larger the font size, the more time it takes for rendering. We will explore other opportunities where the measurement resolution is sufficient (e.g., larger fonts are used). Another direction is to integrate this attack with scheduler-based attacks [53, 63, 109] to slow down the victim process which in turn allows the measurement to be more precise. Finally, is is worth noting that most previous attacks against crypto libraries assume a large number of observations [109] (as the encryption can be triggered by the attacker) which makes their attack much easier compared to ours (from the measurement challenge perspective).

Capital letters and special characters: In reality, many passwords must include capital letters and special characters. Adding these characters directly to our prediction model would no doubt introduce confusion and reduce accuracy. Fortunately, often times these characters can only be entered by pressing special keys (e.g. shift, "?123") or perform special actions (e.g. long-presses). These operations will change the keyboard status (e.g. switch to special characters keyboard) and generate unique signals (e.g. redraw the keyboard) that can potentially be detected by the attacker. Therefore, attacker can train different

prediction models for capital letters and special characters. Upon detecting a keyboard status change, attacker can then switch to the corresponding prediction model.

Mitigations: There are several steps one could take to help mitigate the side-channel attack that we discover. Since the attack relies on the flush+reload attack, disabling user access to the CLFLUSH instruction and high resolution timers will make the attack much more difficult. Although the attacker could still evict a cache line by accessing a set of memory blocks, it will be much slower and result in an attack with much lower resolution. Since performance is critical to our attack, this is likely to reduce the accuracy significantly. Disabling the high-resolution timer will also affect our attack. However, the attacker could choose to implement its own timer [72] and perform the attack as described.

A general solution to timing side-channels is to make the rendering constant time irrespective of the input at the cost of rendering performance. Nevertheless, even if one decides to implement this mitigation, one will need to overcome the challenge of locating the input-dependent subroutines. Here, our profiling model can significantly help in identifying these locations and thus can be useful for defense as well.

Finally, to prevent attacks on applications such as CapitalOne, a user can turn off the "Make password visible" option (which is on by-default) under the Android settings. This makes password inputting less convenient but prevents any text rendering operation for passwords. In addition, application developers can choose to forcibly pre-render all characters with the same font as the password, thus eliminating the font translation process.

Extensions: In this paper, we focus on using our attack to discover side-channels in graphic libraries. In principle, our attack on input-dependent execution times could find previously

unknown side-channels in all kinds of shared libraries. As a future work, we will consider extending our attack beyond graphic libraries such as crypto and audio processing libraries, hardware drivers, etc. In addition, currently we are mainly studying applications on Linux and Android. Our attack could also be ported to platforms such as Windows and MacOS.

Additionally, it's worth noting our attack implicitly obtains the inter-keystroke timing for free via flush+reload. This allows us to combine our attack with existing inter-keystroke timing attacks [92] to further improve its effectiveness, which we will consider in future studies.

Finally, we currently only use our intuition to exploit a general type of feature - measuring the execution time between two addresses. There might exist other type of features (e.g. execution order, time series of multiple addresses, etc.) in the program execution trace that could potentially be identified using more sophisticated techniques such as deep learning. This is another interesting direction for future studies.

4.8 Conclusions

In this chapter, we discover a previously unknown type of potent side-channel that allows an attacker to use the flush+reload attack to perform cross-process timing measurements on sensitive functionalities inside shared graphic libraries. The attack facilitates the inference of a user's keystrokes when the typed keys are rendered on the screen. The attack hinges on utilizing machine learning techniques to discover execution-time based side-channels inside graphic libraries. We have completely automated the discovery of such side-channels and even the generation of exploits. We validate that our attack is

viable on real-world applications on multiple platforms and demonstrate its high accuracy in predicting user input in practice, which affects a large user population of the considered applications. Finally, we suggest ways to mitigate this exploit.

Chapter 5

CPU Side-Channel Feedback for Fuzzing Un-Modifiable Binaries

This chapter documents our initial research of using CPU side-channels to help fuzzing un-modifiable binaries.

5.1 Introduction & Intuition

Grey-box fuzzing is a very effective technique for discovering software vulnerabilities. Instead of requiring full knowledge of the program structure, grey-box fuzzers conduct light-weight instrumentation to fuzz target to extract feedback of each fuzzing input. This not only leads to a reasonable performance overhead comparing to the programming-analysis-based approaches, but also informs the fuzzer about the increase in code coverage during fuzzing. Advanced grey-box fuzzers such as Syzkaller [9] can fuzz software as complex as operating system kernels with impressive efficiency.

Many grey-box fuzzers, Syzkaller included, requires modifications (i.e. instrumentations) to the fuzzing target to gather coverage feedback. However, this is not always possible as sometimes there are protections (e.g. locked bootloader, Trustzone) that prevents modifications to the fuzzing target. Without any feedback, grey-box fuzzers cannot gain any information about code coverage. Thus, their effectiveness is severely limited.

CPU side-channels have proven to be very effective in breaking many types of isolation techniques. It's very common for CPU side-channel attacks to extract sensitive information of a higher-privilege process (e.g. kernel, secure application) from a lower-privilege process (e.g. user process, loadable kernel module). Therefore, we propose to leverage CPU side-channels as a source of code coverage feedback when target cannot be directly modified. Our goal is to fuzz un-modifiable (i.e. un-instrument-able) binaries in a grey-box fashion, thus improving fuzzing performance.

We consider the following fuzzing scenarios:

1. **Android kernel**. Most Android phones nowadays come with a locked bootloader to prevent modifications to the system or kernel image. Although some manufacturers allow users and third-party developers to unlock the bootloader, manufacturers such as Huawei [16, 17] do not provide any means to unlock the bootloader. Therefore, fuzzing techniques such as Syzkaller cannot take advantage of feedback from an instrumented kernel. In this scenario, it's often impossible to root the device. Therefore, our fuzzer can only have userspace privilege.

2. **SGX / Trustzone applications**. Intel SGX and ARM Trustzone are designed to host secure applications in enclaves that are immune to normal attacks from an

insecure world. In the case of Trustzone, manufacturers often deploy trusted OS, boot, and firmware. These secure applications are signed by the manufacturer and cannot be modified to provide fuzzing feedback. In this scenario, we assume our fuzzer has non-secure kernel privilege.

Depending on the different scenarios, we expect to face different technical challenges. In general, these technical challenges can be categorized as follows:

1. **Reliable side-channel**. Similar to researches conducted in previous chapters, the quality of the side-channel we're using determines our capacity. If the side-channel is noisy and unstable, we might need to run the same input repeatedly for a large number of times in order to obtain a clear signal. This defeats the goal of improving performance over blackbox fuzzing. For example, on some Android devices, prime+probe attack from userspace is very unreliable due to lack of pagemap access and ARM's random replacement policy. On the other hand, the RAMINDEX operation [1] from kernel space provides a much more stable signal.

2. **Interface Knowledge**. It is essential we have the knowledge of the interaction interface between our fuzzer and fuzzing target. This is not a problem when fuzzing kernel as the system call interfaces are open and previous kernel fuzzers (e.g. Syzkaller) had established well-founded methodologies in generating fuzzing inputs accordingly. However, the interface for secure application and OS are not as well documented. Additionally, there is no effective fuzzer for secure applications to our best knowledge. We will have to build a fuzzer from scratch.

```
#define __SYSCALL_DEFINEx(x, name, ...)                 \
    asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__)); \
    static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__)); \
    asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__))   \
    {                                   \
        long ret;                       \
        ramindex_prime(); /* Occupy L1 Instruction Cache */ \
        ret = SYSC##name(__MAP(x,__SC_CAST,__VA_ARGS__));   \
        ramindex_probe(); /* Probe L1 Instruction Cache */ \
        __MAP(x,__SC_TEST,__VA_ARGS__);             \
        __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));   \
        return ret;                     \
    }                                   \
    SYSCALL_ALIAS(sys##name, SyS##name);            \
    static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__))

#else
```

Figure 5.1: Implementing RAMINDEX prime+probe attack in Linux kernel.

5.2 Evaluation: Side-channel Quality

In order to address the first challenge we identified in Section 5.1, we first evaluate the possibility of using CPU side-channel as feedback for fuzzing. Specifically, we use the RAMINDEX interface, which allows OS kernel to directly read CPU cache content, as the most noise-free form of prime+probe. We evaluate whether CPU prime+probe side-channel in its most noise-free form could serve as an effective source of coverage feedback for the state-or-art fuzzer such as Syzkaller.

We modified the kernel version 3.10.73 for LG Nexus 6P phone running Android 8.0. First, we back-ported the Kcov [20] interface to this kernel to allow Syzkaller to obtain coverage information. Next, we implemented the L1 instruction cache prime (occupying the entire L1 instruction cache) and probe (checking the entire L1 instruction cache for replacement and output to Kcov interface) functions using RAMINDEX interface. We choose to monitor L1 instruction cache as it closely resembles the code being executed

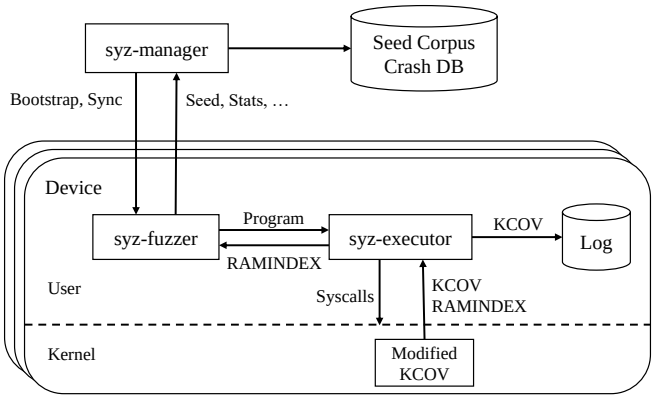


Figure 5.2: Evaluation framework: modified Syzkaller.

during a system call. L2 cache on the Nexus 6P is unified and shared among two CPU cores, which will no doubt introduce noise due to data accesses and activities from other processes. Finally, we modified the system call interface as illustrated in Figure 5.1, priming L1 instruction before the core system call functionalities and probing L1 instruction cache afterwards. We also modified Syzkaller as depicted in Figure 5.2, allowing it to operate with feedback from RAMINDEX prime+probe on the L1 instruction cache. We also collect the regular Kcov-produced branch coverage as the ground truth for comparison.

On the LG Nexus 6P phone, we performed 6-hour fuzzing experiments using three different setups of Syzkaller: 1) the default Syzkaller using regular Kcov-produced branch coverage, 2) the modified Syzkaller using RAMINDEX prime+probe result as coverage and 3) the black-box Syzkaller that received no coverage information and can only generate inputs blindly. The result of our experiment is shown in Figure 5.3.

As we expected, RAMINDEX prime+probe is an inferior source of feedback comparing to regular branch coverage. Despite being the most noise-free form of prime+probe

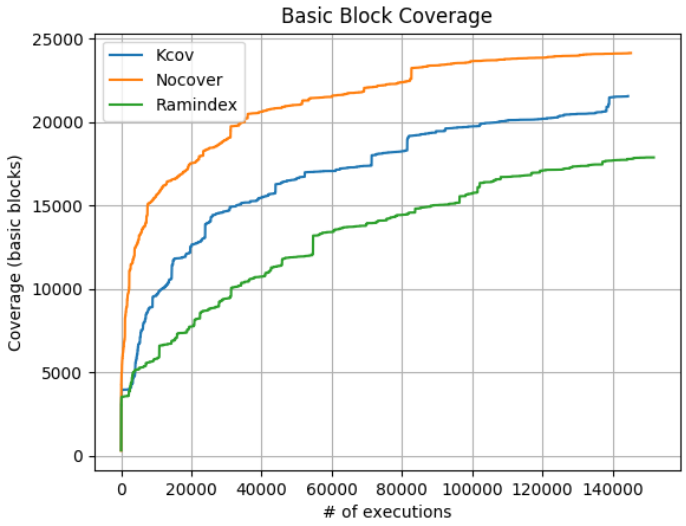


Figure 5.3: Comparison between KCOV, RAMINDEX and generation-only (No cover) Syzkallers.

we can implement, there could still be noise due to prefetching, speculative execution, etc. In addition, L1 instruction cache prime+probe is limited to the 64-byte cache-line size granularity, which in some cases are insufficient for identifying branches.

What interests us most is that both RAMINDEX and the default Syzkaller is out-performed by the Syzkaller with no coverage at all. This is very counter-intuitive since grey-box fuzzing is supposed to a significant improvement over black-box fuzzing. Further investigation reveals the cause of our observation:

1. When generating input from scratch, Syzkaller utilizes a builtin library of `templates`. These templates are manually curated by domain experts (e.g., kernel developers), and contain information relating to the argument type of each system call, and the dependencies between system calls These well-written makes generating new inputs very effective, especially in the early stages when the kernel is not explored much.

2. Due to the stateful nature of OS kernels (e.g., certain code may be allowed to execute only once, controlled by a global flag). Syzkaller needs to perform "`triage`" on any new programs discovering new coverage. This process includes "`verification`" by re-executing the program and verify the new coverage can be reliably reproduced, as well as "`minimization`" by attempting the removal of system calls and/or the shortening of the arguments, while still retaining new coverage. This process takes a long time (comparing to generating a program) and does not produce much new coverage by itself.

As the fuzzer runs for longer, these problems will eventually fade away and the default Syzkaller is able to reach more coverage than the black-box generation-only Syzkaller, as we will show in Section 6.2.2. However, on systems with limited throughput such as Android phones, it would take significantly longer for the default Syzkaller to outperform generation-only Syzkaller. After carefully studying Syzkaller's strategies, including the priority of different tasks and seed selection strategy (see Section 6.2.1), we believe there are plenty of opportunities for improvement and the coverage growth of Syzkaller can be significantly increased. We hence propose to make these improvements using reinforcement-learning techniques, as we will introduce in Chapter 6.

5.3 Conclusion

In conclusion, we present the idea to use CPU side-channel as a feedback source for grey-box fuzzing. We evaluated RAMINDEX prime+probe, a more noise-free version of the prime+probe, on Syzkaller and measured its effectiveness when fuzzing Android kernel.

Surprisingly, we discover that there is an opportunity of improvement on default Syzkaller's strategy in the early stages of fuzzing, especially on devices with limited throughput. In the end, we are able to turn this idea into full-fledged research on dynamically adjusting kernel fuzzing strategies with reinforcement learning, which will be presented in detail in Chapter 6.

Chapter 6

SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning

This chapter documents our research of improving kernel fuzzing efficiency with reinforcement learning.

6.1 Introduction

Gray-box fuzzing or *coverage-guided fuzzing*, is an automated software testing technique that has gained traction in recent years. In a nutshell, such a *fuzzer* automatically provides unexpected, or random data as inputs to explore the codebase, to maximize code coverage and/or reveal bugs. Although fuzzing is generally effective in practice, it is often perceived as an art, as every fuzzer embeds various heuristics throughout the process. More specifically, fuzzers often have many decision points and parameters (e.g., which seed to mutate) that collectively determine their overall effectiveness. Making these choices often

involves not only strong intuitions and domain expertise, but also much empirical testing and tuning.

Even though researchers have been attempting to auto-tune various fuzzing decisions, including seed selection [103, 110, 87] and mutation operators [31, 61, 30, 74], prior efforts are mostly point solutions and none of them are specifically tailored for Operating System (OS) kernel fuzzing. Kernel fuzzing is uniquely challenging for the following reasons: (1) any modern OS kernel has a huge code base and dependencies among various components; (2) the input to an OS kernel is through the system call interface that needs special handling; and (3) an OS kernel maintains a massive amount of state and the behavior of an input (i.e., test case) may or may not be reproducible. To illustrate these factors, the state-of-the-art kernel fuzzer, Syzkaller [9] itself has over 1.3 million lines of code and numerous parameters than can be tuned to improve its efficiency. Given this large and complex space, and the ad hoc strategies used to tune parameters, we believe that there remains an abundance of opportunities to improve kernel fuzzing.

To address the above OS kernel fuzzing challenges, Syzkaller employs a combination of generation-based [43] and mutation-based [18] input crafting strategies. Specifically, to generate inputs (sequence of syscalls) from scratch, Syzkaller requires hand-crafted input models called "templates". It also leverages mutation to take known good inputs (aka. *corpus seeds*) that have previously achieved new code coverage, and mutate (i.e., modify) them to generate new ones. Finally, Syzkaller needs to triage an input to make sure that a minimal input can reproduce the coverage that it achieved before turning it into a seed. Syzkaller uses a fixed strategy to schedule these different types of tasks and seeds to mutate.

In this paper, we propose SyzVegas, a fuzzer based on Syzkaller that is capable of dynamically and automatically adapting its strategies to improve coverage. Specifically, we focus on addressing the two aforementioned first-order decision making processes: 1) selecting (scheduling) the most rewarding fuzzing tasks (e.g., generation, mutation and triage) and 2) selecting the most potent seeds for mutation. Both of these are done dynamically in SyzVegas via a unified reward assessment model to significantly improve the odds of excavating new code coverage and finding new vulnerabilities.

The main contributions of our paper are as follows:

- **Identifying optimization opportunities.** We perform a systematic analysis of Syzkaller's default (fixed) task and seed selection policies. We identify several opportunities for improving Syzkaller's fuzzing efficiency.

- **Realizing dynamic fuzzing.** SyzVegas employs a light-weight reinforcement learning algorithm to adjust the task and seed selection policies dynamically. We propose a novel approach for modeling the rewards attained by the different fuzzing tasks by consolidating the discovery of new coverage and the time cost incurred. The approach also accounts for the associations between different types of tasks, can quickly adapt during the different stages of fuzzing, and has very low overhead.

- **Improved coverage growth.** We perform extensive evaluations of SyzVegas on the latest Linux kernel and show that it consistently attains 21.6% more coverage than the default Syzkaller and finds more unique crashes. In total, we found 11 crashes that the default Syzkaller failed to detect in the same time period. For OS kernels as important as Linux, such an improvement makes a big difference as every kernel

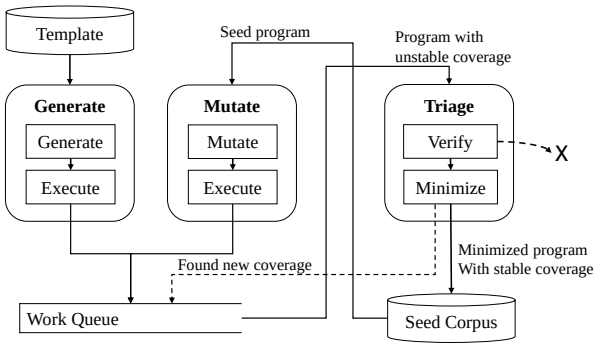


Figure 6.1: Workflow overview of Syzkaller.

version is being constantly fuzzed and tested (e.g., by Google [45] using hundreds of machines.

6.2 Background and Motivation

6.2.1 Syzkaller

The goal of Syzkaller is to explore the OS kernel by executing a series of test **programs**, defined as a sequence of system calls. To create such programs, Syzkaller has two options: generate a new program from scratch or mutate from an existing program. Figure 6.1 depicts the workflow of Syzkaller. It has three types of tasks during the fuzzing process viz., **Generation**, **Mutation** and **Triage**.

- **Generation**. Syzkaller creates a brand new test program using *templates*. These templates are manually curated by domain experts (e.g., kernel developers), and contain information relating to the argument type of each system call, and the dependencies between system calls (e.g., the return value of `open` can be used later in `read`).

This allows Syzkaller to generate meaningful syscall sequences and arguments, greatly improving the likelihood of exploring deeper kernel code. Generation is a highly independent task and does not rely on other types of tasks. If a generated program produces new coverage, it is then put into the triage *work queue*.

- **Mutation**. Syzkaller randomly picks a program (a.k.a. *seed*) from a *corpus* (i.e., programs that have previously achieved new coverage), and performs a series of random mutations on the chosen seed (e.g., inserting/removing a new syscall, changing the argument of an existing syscall), and then execute it. Similar to generation, if a mutated program produces new coverage, it is inserted into the triage work queue.

- **Triage**. Syzkaller fetches a program from the triage work queue. The picked program is in the queue because it was observed to attain new coverage. However, at this point it is unclear whether the coverage can be reproduced reliably, due to the stateful nature of OS kernels (e.g., certain code may be allowed to execute only once, controlled by a global flag). Therefore, Syzkaller first performs "**Verification**" by re-executing the program thrice and computing the coverage that is stable throughout the re-execution. If the stable coverage is empty (none is attained), then the triage is aborted. Otherwise, Syzkaller performs a "**Minimization**" of the program by attempting the removal of system calls and/or the shortening of the arguments, while still retaining the stable coverage. Finally, Syzkaller puts the minimized program into the seed corpus (where further mutations can be performed later). During the minimization, Syzkaller might discover that a partially minimized program can achieve new coverage; in such cases, Syzkaller will put these programs into the work queue to be triaged later.

By default, Syzkaller selects the aforementioned three types of fuzzing tasks as per the following hard-coded priorities:

1. Triage takes absolute priority over generation and mutation.

2. When no triage task is available, the absolute priority goes to mutating programs that were just added to the seed corpus. Syzkaller will mutate each new seed for a fixed number of (100) times. These mutations receive some special treatment and are referred by Syzkaller as **Smash**.

3. If no triage or smash tasks are available, Syzkaller will execute generation and regular mutation tasks with a fixed 1:99 ratio, i.e., one generation task for every 99 mutation tasks.

In practice, when Syzkaller starts from scratch, a generation task is performed first and some part of the kernel code base is covered as a consequence. This very first program will then go through triage, producing the initial seed and potentially creating more programs for triage during minimization. Then, Syzkaller will focus on triaging these additional programs (if any from minimization) and smashing the new seeds, which in turn creates more seeds for smashing and programs for triaging; proceeding in this manner typically leads to a huge chain reaction. As a result, the actual number of generations Syzkaller performs is much lower than the policy description may suggest.

When it comes to mutation, Syzkaller chooses which seed to mutate according to the following principles. First, as mentioned before, a newly created seed enjoys the privilege of a high-priority invocation of 100 mutations, i.e., smash. Second, each seed has

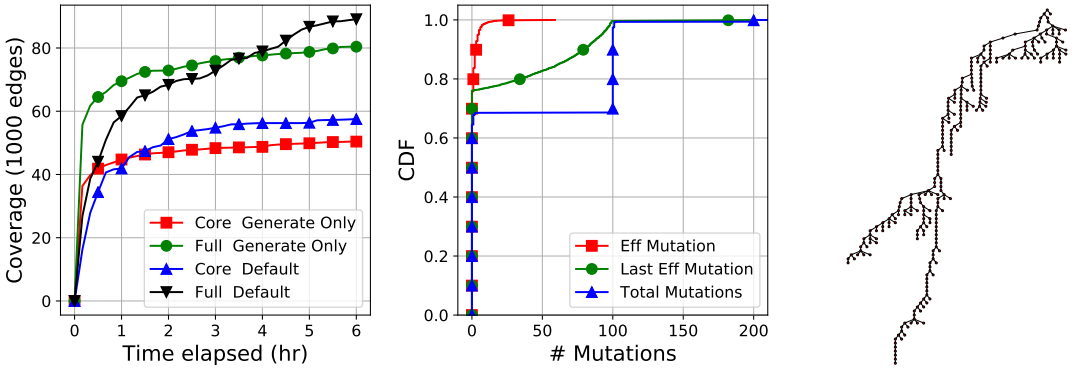
a weight assigned to it equaling the number of new and stable edge coverage it brings. This number is static and will not change over time. When Syzkaller needs to pick one seed from the corpus, it will do so on the basis of this weight, from among all the seeds.

6.2.2 Observations and Intuition

In this section, we motivate the rationale of using a learning-based approach to improve Syzkaller's coverage. We run the default Syzkaller alongside a modified Syzkaller which can perform only generations, for 6 hours, and collect metrics such as coverage growth and task effectiveness to gain insights about the operation of Syzkaller. We use a testbed with Intel(R) Xeon(R) CPU E5-2680 v4 2.40 GHz CPU as the testing platform. Both the default Syzkaller and the generation-only Syzkaller run on a 2-core single-process fuzzer VM. Our experiment yields the following observations:

The best strategy evolves over time. Following the task selection policy described earlier, Syzkaller gives a low priority to generation. However, with a well-written template, generation can be quite powerful, especially in the earlier stages of the fuzzing where most of the kernel code is unexplored/uncovered.

Figure 6.2(a) demonstrates the coverage growth comparison between the two Syzkallers fuzzing 1) the full Linux kernel and 2) the core kernel excluding sub-systems such as filesystem and drivers. When it comes to fuzzing the full Linux kernel, we observe that in the first 1 hour of fuzzing, the generation-only Syzkaller outperforms the default Syzkaller by a significant margin. After 4 hours, however, the generation-only Syzkaller falls behind the one using the default strategy. When fuzzing the core kernel, however, it



(a) Coverage growth comparison (b) Mutation effectiveness (c) Sampled program evolution

Figure 6.2: Evaluating default Syzkaller strategies.

takes only 1 hour for the default Syzkaller to find more coverage than the generation-only Syzkaller. These observations suggest that the optimal strategy is dynamic, and should adapt over time in a way that is sensitive to the state of the fuzzer and fuzzed kernel. Although we tried multiple static strategies (e.g., generate more earlier), they are not able to effectively adapt and outperform the default Syzkaller. This motivates a learning-based approach.

Ad-hoc decisions can be harmful. Syzkaller's strategy gives high priority for mutating newly-discovered seeds, invoking a mandatory 100 mutations, to extract as much as possible quickly from a fresh seed. There is little doubt that the domain experts working on Syzkaller choose this strategy carefully with extensive testing. However, this ad-hoc decision leaves opportunities for improvement. Figure 6.2(b) shows the mutation effectiveness of the default Syzkaller, fuzzing the whole Linux kernel, for 6 hours. We see that there are three opportunities for improvement: 1) A large number of seeds are not being mutated

Algorithm 2 *Exp3-IX* Algorithm

$w_i \leftarrow 0$. for $i = 1, ..., K$

for all $t = 1, 1, 2$ **do**

  $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}, \quad$ for $i = 1, ..., K$

  Draw i_t randomly according to $pr_i(t)$

  Receive reward $x_{i_t}(t) \in [0, 1]$

  for all $i = 1, ..., K$ **do**

$$\hat{x}_i(t) = \begin{cases} x_{i_t}(t)/(pr(i) + \gamma), & i = i_t \\ 0, & \text{otherwise} \end{cases}$$

    $w_i(t + 1) = w_i(t) \cdot e^{\eta \hat{x}_i(t)}$

  end for

end for

because Syzkaller is too busy performing the mandatory number of mutations and triaging. 2) We observe chain reactions where the 100 new mutations of a program discover new coverage and in turn schedule additional 100 new mutations for each of these, causing exploration to be focused narrowly on the seeds from the same roots; and, 3) Of those seeds that are mandatorily mutated, many do not deserve to be mutated 100 times. We believe that this behavior is an unintended consequence of the ad-hoc (but perhaps empirically acceptable) decision to carry out 100 mutations of each new seed. We also believe that a learning-based approach can avoid these negative consequences and therefore improve the fuzzing effectiveness.

Kernel-space must be effectively explored. In Syzkaller, all explorations start from generated programs. These generated programs then go through a series of minimization and mutation tasks, creating a forest-like structure with a tree rooted at each generated

program. Essentially, kernel fuzzing can be viewed visually as a tree/forest exploration, where Syzkaller attempts to explore/grow the program evolution trees in order to find new coverage. There are multiple strategies to approach this problem. For example, how many roots (or trees) should the fuzzer "plant", should the fuzzer favor breadth-first or depth-first approaches, etc.

Figure 6.2(c) shows a down-sampled program evolution tree from fuzzing the full kernel for 24 hours. The default Syzkaller plants very few trees due to its very low priority of generation. It also heavily favors a depth exploration approach of these few trees owing to its high priority to mutation, especially the mandatory 100 mutations for new seeds (which are not always effective). Our intuition (which we back experimentally later) is that each tree can only cover a limited part of the kernel, and a strategy that explores the kernel space well must learn how to find the interesting trees and how to grow them.

Intuition. Based on the observations above, we conclude there are plenty of opportunities to tune the various hard-coded parameters (e.g. mutation count, generation to mutation ratio) and priorities. Our experiments suggest that the right strategy and the right seed dynamically change over time. Needed is an automated way to identify the task that is most promising at any given time, and if appropriate the best seed to be invoked in association with that task. To identify the best task and the best seed, a *reinforcement-learning* based scheme is a natural fit, where it can be modeled to maximize the coverage rewards relative to the time cost of execution.

6.2.3 Multi-armed Bandit Problem

The *Multi-Armed Bandit* (MAB) problem is a reinforcement learning problem, which we believe is well suited to model the various decisions of Syzkaller. In this problem, a gambler must play a number of competing slot machine arms (choices) in a way that maximizes their expected gain. Each arm's properties are only partially known at the time of playing and may become better understood as the arm is played more and more. The MAB problem is a classic example of the tradeoff between *exploration and exploitation*.

While there exist a multitude of reinforcement learning models and algorithms, we consider the MAB problem to be particularly suitable because it is non-associative [93], meaning that it passively adapts to the changing reward signals without explicitly concerning itself with the long term implications of taking an action. As a result, although hard to achieve the absolute optimal strategy, it has the advantage of learning the dynamics much quicker and adapting faster. In addition, it is computationally efficient to implement, which is critical in maintaining the throughput in fuzzing.

One notable variant of the MAB problem is called *Adversarial Bandit* problem, first introduced by Auer and Cesa-Bianchi in 1995 [27]. In this variant, the slot-machines are controlled by an adversary who is capable of altering the reward of the arms every play. As one of the strongest generalizations of the MAB problem, the adversarial bandit problem requires its solution to react quickly to the changing rewards of each arm. This maps well to the fuzzing process where each decision also receives different reward over time.

To address the adversarial bandit problem, Auer et.al. proposed the *Exponential-weight algorithm for Exploration and Exploitation* (*Exp3*) [28]. The main idea is to introduce

an exponential growth in an arm's weight (i.e. probability of playing) depending on the yielded reward, thereby ensuring that good arms are quickly identified and exploited. There are many variants of the *Exp3* algorithm. *Exp3.1* [28] divides the algorithm executions into epochs and resets *Exp3* at the beginning of each epoch, making *Exp3* perform better over time. *Exp4* [28] allows for an additional advice vector to be inputted. *Exp3-M.B* [111] extends *Exp3* to playing multiple arms at the same time with a limited budget. *Exp3-IX* [77] replaces the explicit exploration with implicit exploration, further improving its regret bounds.

Algorithm 2 shows the *Exp3-IX* algorithm. *Exp3-IX* maintains the weight of each of the K arms, each of which is used to proportionally determine the playing probability of that arm. When an arm is played and a reward is attained, the algorithm first computes the estimated reward based on the probability of playing this arm as well as an implicit exploration factor γ. The weight of each arm is increased exponentially based on the estimated reward, controlled by the constant growth factor η. Given the number of arms K, the total number of plays T, and an exploration/growth factor $\eta = 2\,\gamma = \sqrt{\frac{2\,ln\,K}{KT}}$, *Exp3-IX* guarantees a regret bound of:

$$G_{max} - E(G_{Exp3-IX}) = \sqrt{2\,KT\,ln\,K} + \left(\sqrt{\frac{2\,KT}{ln\,K}} + 1\right) ln\,\left(\frac{2}{\delta}\right) \tag{6.1}$$

with probability of at least $1 - \delta$ for any $0 < \delta < 1$.

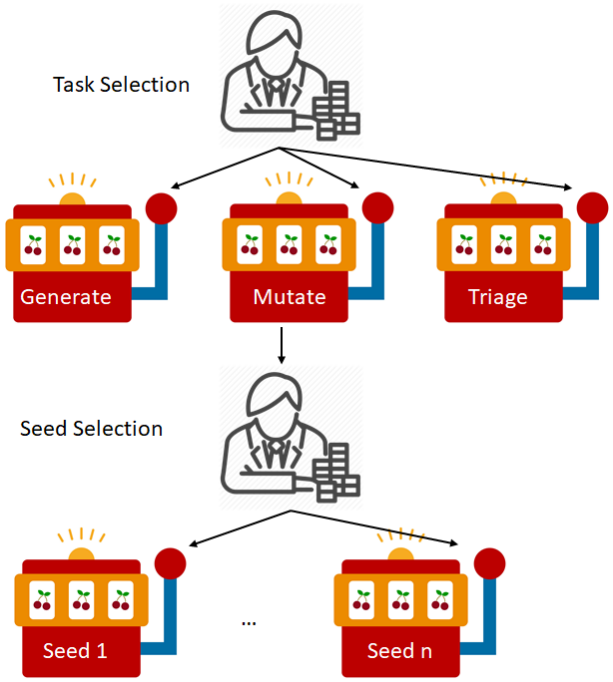


Figure 6.3: High-level idea/design.

6.3 Design and Implementation

We propose SyzVegas, a dynamic fuzzing approach to select between the three types of tasks in Syzkaller. The main design goals of SyzVegas are as follows:

- *Optimal coverage.* SyzVegas should select tasks or pick a mutation seed program in such a way that maximizes the coverage achieved by Syzkaller while minimizing the incurred time cost.

- *Adaptive adjustment.* SyzVegas should determine which type of task is the best, at any (and every) stage of fuzzing, and adapt its strategy accordingly. When performing mutations, SyzVegas should be able to assess the quality (change) of the mutated seed and adjust its weight in the seed corpus accordingly.

To achieve these goals, SyzVegas abstracts the task selection problem as an Adversarial Multi-armed-bandit (MAB) problem, where generation, mutation and triage are treated as the three separate arms. When performing mutation, we treat seed selection as another layer of the MAB problem, i.e., each seed is treated as a separate arm. In each layer, SyzVegas utilizes an algorithm similar to classic Adversarial Bandit problem solutions such as *Exp3-IX* [77] and *Exp3.1* [28], to make decisions on which "arm" to "play" towards maximizing the reward over time. Figure 6.3 illustrates the high-level idea/design of our system.

Referring back to Figure 6.1, Syzkaller collects information relating to two types of coverage, viz., unstable coverage and stable coverage. We design SyzVegas to optimize for maximum unstable coverage; this is because both types of coverage can lead to crashes but unstable coverage is a superset of the stable coverage.

Given that SyzVegas treats the task selection and seed selection as Adversarial MAB problems, the key challenges we need to address are: 1) how to assess the value of the selected task or the mutated seed, 2) how to pick the task or seed that has the maximum potential. We address how SyzVegas overcomes these challenges in this section. Table 6.1 lists the symbols we use in subsequent sections to allow the reader to quickly find associated definitions.

6.3.1 Reward Assessment

To begin with, whenever a generation/mutation/triage task has completed execution, we need to assign a reward to the task; this reward is later be used when we use

Table 6.1: Symbols we use to describe SyzVegas

Symbol	Description
c or c_i	Number of edge coverage attained by executing a single task (of type i).
t or t_i	Execution time of a single task (of type i).
C	Total edge coverage attained throughout fuzzing.
T	Total elapsed time of fuzzing.
t_{exp}	Estimated expected execution time of task/tasks.
g or g_i	Un-normalized reward attributed to task/tasks (of type i).
$c^p_{mut}(m)$	Total edge coverage of mutating a seed p for m times.
$t^p_{mut}(m)$	Total execution time of mutating a seed p for m times.
x	Normalized reward attributed to task/tasks.
$\hat{G}_i$	Accumulated reward estimation of MAB arm i.

the Adversarial Bandit problem model. The key requirements/challenges in computing this reward are as follows:

1. *Gain and cost considerations.* The goal of SyzVegas is to maximize the gain (i.e. number of edges covered) while minimizing the cost (i.e., the time taken for execution). Our model must be able to unify these two metrics with different units into a single measurement of the effectiveness (utility) of each task.

2. *Dependencies between tasks.* The classic multi-armed-bandit problem assumes that each arm is independent. However, this is clearly not the case in the context of Syzkaller. As shown in Figure 6.1, there is a strong relationship between Triage and Mutation. SyzVegas needs to properly address this relationship when assigning rewards to each arm.

3. *Normalization.* The utilities observed on different systems can be different. For example, the time it takes to execute a program on Android will be much longer than executing the same program on a powerful server. In addition, the algorithms that are used to solve an Adversarial Bandit problem often require the reward to be normalized.

To address the above challenges, we construct our reward assessment model as follows, considering each of the tasks of interest.

Generation. Generation is not directly intertwined with either mutation and triage. Thus, it's reward can be assessed independently. Let c be the new coverage (measured by the number of edges) obtained by generating a program. Let t be the cost in time of executing the generated program. Let C and T be the total achieved coverage (regardless of whether contributed by generation), and the total elapsed time from when the fuzzer began, respectively. Given these, the expected time for finding the new coverage c (given our average performance up to time T), can be "estimated" by:

$$t_{exp} = c \cdot \frac{T}{C} \tag{6.2}$$

The reward for the generation task can be modeled as the expected time cost minus the actual time cost t:

$$g = t_{exp} - t = c \cdot \frac{T}{C} - t \tag{6.3}$$

Note that g essentially draws a comparison between the coverage discovery rate of the current generation task c/t and the coverage discovery rate historically C/T. If the task has a better-than-historic coverage discovery rate, it will always have a positive reward, while a worse-than-historic coverage discovery rate will result in a negative reward. This

reward representation also ensures that if two tasks A and B both produce the same coverage c, but consume different times, say $t_A > t_B$, we always have $g_A < g_B$. This is intuitive since a task that discovers coverage at a higher rate should be rewarded more. Note that we use time instead of rate as the unit of the reward. This ensures that if two tasks A and B both produce no new coverage (which happens often in later stages of fuzzing), we always have $g_A < g_B < 0$. In other words, a task that wastes more time will be punished harder than a task that wastes less time.

Mutation and Triage. Mutation tasks are heavily dependent on triage because: 1) the seed driving a mutation can only be obtained via triage and 2) triage will try to minimize the seed, thus reducing costs for future mutations. As a result, the reward of mutation and triage must be modeled in conjunction.

Consider a seed program p, where the cost of the triage task that verified and minimized p is denoted as t^p_{tri}. As discussed in section 6.2, triage consists of two phases viz., verification and minimization. The time costs of each phase are denoted as t^p_{ver} and t^p_{min}, with $t^p_{ver} + t^p_{min} = t^p_{tri}$. During the minimization, triage first receives a generated/mutated program p' and "minimizes" it to p by removing system calls and/or shortening arguments. Let $t^{p'}$ and t^p denote the costs of executing the programs p' and p, respectively. Thus, the time saved due to minimization is $\Delta^p_t = t^{p'} - t^p$. Minimization can also potentially discover new coverage which will be triaged later. We denote the coverage achieved due to minimization as c^p_{min}.

In reality, the verification phase can also produce new coverage from simply re-executing the original program. However, since this new coverage was not observed in the

previous execution of the same program, the input program in this form is unstable (the coverage may not be reproducible) by definition. As a result, Syzkaller does not attempt to process such new coverage possibilities. We follow Syzkaller's design on this matter i.e., ignore new coverage possibilities from verification.

The seed program p, is then mutated m times. The observed edge coverage with each mutation are $c_1^p, c_2^p, ...c_m^p$, while the time costs associated with each of these mutations are $t_1^p, t_2^p, ..., t_m^p$, respectively. For simplicity, we denote:

$$c_{mut}^p(m) = \sum_{j=1}^{m} c_j^p \quad , \quad t_{mut}^p(m) = \sum_{j=1}^{m} t_j^p. \tag{6.4}$$

Note here that without minimization, Syzkaller can only mutate from p' instead of p. In this case, on average, each mutation should take Δ_t^p longer; thus, minimization results in a total of $m \cdot \Delta_t^p$ time savings, over m mutation tasks. If we treat the one triage and m mutations as a single task, we can then compute the expected time to discover the new coverage of $c_{mut}(m)$ without minimization as:

$$t_{exp}^p = (c_{min}^p + c_{mut}^p(m)) \cdot \frac{T}{C} + m \cdot \Delta_t^p \tag{6.5}$$

The first part of right hand side of the equation, is an estimation of the total expected time to discover the new coverage $c_{min}^p + c_{mut}^p(m)$ by mutating p; the second part of the equation is the estimated time savings due to minimization. The total reward from triaging and mutating seed p is the difference between the "expected and actual time", given by:

$$g_{tri+mut}^p = (c_{min}^p + c_{mut}^p(m)) \cdot \frac{T}{C} + m \cdot \Delta_t^p - (t_{tri}^p + t_{mut}^p(m)) \tag{6.6}$$

Since triage and mutation take different amounts of time, we need to distribute the reward proportionally to each arm. In addition, since triage has two phases (with different purposes), the reward should be attributed to each phase separately.

We reiterate here that since the main contribution of minimization is to save time in future mutations, the time savings part of Equation 6.6 must be fully credited to minimization. In addition, minimization is also finding new coverage c_{min} from testing minimized programs. Combining them both, we can thus estimate the reward attributed to minimization as:

$$g^p_{min} = c^p_{min} \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{min} \tag{6.7}$$

Verification is essential for creating the seed p, which is later mutated m times to obtain new coverage. Without verification, mutation will have no seeds to mutate and therefore useless. Thus, verification and mutation should share the reward of obtaining new coverage proportional to their costs. As a result, the reward attributed to verification and mutation are:

$$g^p_{ver} = c^p_{mut}(m) \cdot \frac{t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{ver} \tag{6.8}$$

$$g^p_{mut} = c^p_{mut}(m) \cdot \frac{t^p_{mut}(m)}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{mut}(m) \tag{6.9}$$

Adding Equation 6.7 and Equation 6.8, we obtain the total reward attributed to triage as:

$$g^p_{tri} = \left(\frac{c^p_{mut}(m) \cdot t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} + c^p_{min} \right) \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{tri} \tag{6.10}$$

Note that Equation 6.9 and Equation 6.10 are only approximate estimates of the rewards with mutation and triage, respectively. In practice, it is difficult if not impossible to predict how many times a seed program p will be mutated. In addition, it is impractical

to compute the reward after all mutations are complete. Every time a seed program p is mutated, we need to update the weight of the triage and mutation arms. To achieve this goal, we first compute the reward for triage and mutation when seed p is added to the corpus via triage as: $g^p_{tri}(0) = c^p_{min} \cdot \frac{T}{C} - t^p_{tri}$ (as the reward of performing the triage task alone) and $g^p_{mut}(0) = 0$. As p is mutated, we keep track of the observed new coverage and time costs.

Updating rewards. For the k^{th} mutation, we compute the estimated total reward $g^p_{tri}(k)$ and $g^p_{mut}(k)$ using Equation 6.10 and Equation 6.9. We then compute the difference as compared to the estimated total reward after the previous mutation step ($k-1$th mutation) as $\Delta(g^p_{tri}, k) = g^p_{tri}(k) - g^p_{tri}(k-1)$ and $\Delta(g^p_{mut}, k) = g^p_{mut}(k) - g^p_{mut}(k-1)$. We then use $\Delta(g^p_{tri}, k)$ and $\Delta(g^p_{mut}, k)$ as the reward for the triage and mutation tasks at the kth mutation; this is used later in our task selection algorithm (Section 6.3.2).

Normalization. The rewards g for generation, mutation and triage tasks can take values from $(-\infty, \infty)$. However, single-factor algorithms such as Exp3, Exp3.1 and Exp3-IX require the reward be normalized to $[0, 1]$. For budget-constrained algorithms such as Exp3-M.B. [111], both the gain and cost are normalized to $[0, 1]$, and the resulting (gain - cost) belongs to the range $[-1, 1]$. The *Logistic function* $1/\left(1 + e^{-y}\right)$ is a common normalization technique for realizing a normalization from $(-\infty, \infty)$ to $(0, 1)$ [96]. We rescale the logistic function from $(0, 1)$ to $(-1, 1)$ as $\left(1 - e^{-y}\right) / \left(1 + e^{-y}\right)$, ensuring that a zero reward is always normalized to 0. In order to account for the variations, we use $z' = g/\sigma_g$, a shifted version of standard Z-score to replace the y in the logistic function. We shift $z = \left(g - \overline{g}\right)/\sigma_g$, the standard Z-score with a mean of $\overline{g}$ to a mean of 0, in order to make sure that a positive re-

ward g will always be normalized to a positive normalized reward x. The final normalization equation is:

$$x = \frac{1 - e^{-g/\sigma_g}}{1 + e^{-g/\sigma_g}} \tag{6.11}$$

Combined with the gain/cost model, this normalization technique has the following benefits:

1. $g > 0 \Leftrightarrow x > 0$. This means that a task that takes less time than what is expected based on history, will always have a positive gain; a task that takes longer than expected historically, will always have a negative gain.

2. If a task produces no coverage at all, $x < 0$. Intuitively, we do not want to give any positive reward to tasks that only waste time.

3. If two tasks A and B both produce no coverage, but incur different time costs, say $t_A > t_B$, we always have $g_A < g_B < 0$, and $x_A < x_B < 0$. This is also intuitive since a task that waste more time should be punished harder than a task that wastes less time.

6.3.2 Task Selection

Now that we have our reward functions, we design our task selection algorithm based on *Exp3.1* [28] and *Exp3-IX* [77] to determine which task of Syzkaller to invoke at any given stage. We incorporate the exponential weight growth mechanism and the implicit exploration of *Exp3-IX* to ensure sufficient exploitation of the good arms and rapid adaption to changing rewards with regards to the different arms. We combine it with *Exp3.1* that

periodically resets the weight of each arm and adjusts the exploration and growth factor, ensuring the stability of the algorithm over an extended (infinite) period of time. Finally, we combine these with our novel reward assessment model described in Section 6.3.1 to address the association of mutation and triage tasks. The task selection algorithm used in SyzVegas is shown as Algorithm 3.

Similar to *Exp3.1*, the algorithm divides the entire fuzzing timeline into epochs (which is automatically determined by Algorithm 3), indexed by r. Epochs determine when to reset the weights of the arms (required in Exp3.1). For each epoch, our algorithm estimates a target reward $\hat{G}_{threshold}$ for that epoch and tunes the exploration/growth factors γ and η in the same fashion as *Exp3.1*. Within each epoch, our algorithm performs arm selection and reward updates similar to *Exp3-IX*. Upon each update, our algorithm detects if the estimated gain $\hat{G}_i$ exceeds the threshold. If so, the algorithm transitions to the next epoch resetting the observed gains $\hat{G}_i$s to zero and increasing $\hat{G}_{threshold}$ by $4\times$ (for the next epoch).

One major difference between a traditional multi-armed bandit solution and SyzVegas is that we introduce the division of the reward between the triage and mutation functions. The *Exp3* algorithms assume that each arm is independent of each other. As a result, every time one arm is pulled, only the weight of the pulled arm is affected. However, as described in subsection 6.3.1, when the mutation arm is pulled, the weight of both the mutation and triage arms are updated.

Another difference between SyzVegas and *Exp3*-like algorithms is that the normalized reward x_i lies in $(-1, 1)$ in SyzVegas; however, for *Exp3*-like algorithms, the

rewards are often normalized to $[0, 1]$. As discussed in subsection 6.3.1, we made this design choice for two intuitive reasons: 1) we do not want the arms relating to tasks that produce no coverage to receive any gain in weight and 2) when comparing tasks that produce no coverage, we want to punish those tasks that cost more harder. A $[0, 1]$ normalization can only achieve (1) i.e., if tasks produce no coverage at all, they get a reward of 0 regardless of the time cost, but then it cannot achieve (2). Therefore we choose a $(-1, 1)$ normalization instead.

6.3.3 Seed Selection

In addition to choosing the right task (generation versus mutation versus triage), we need to choose the proper seed to be associated with the task in the case of mutation. For this purpose, we again use an *Exp3-IX*-like algorithm which is shown in Algorithm 4. At a high level, the seed selection algorithm inherits the basic ideas of the task selection algorithm. Specifically it includes a reward assessment model, a normalization technique for scaling the reward, and a weight update process. There are some notable differences:

The reward assessment model only considers mutation tasks. When a mutation task is finished, we reuse the gain/loss model discussed in Section 6.3.1 to compute the reward of mutating the current seed. However, since the seed selection algorithm only focuses on mutation tasks, we no longer need to split the reward with triage (as we did with task selection). Instead, we can compute the reward in the same way as Equations 6.2 and 6.3. Moreover, we no longer need to consider the reward created by generation and triage, when it comes to normalization.

Let C_{mut} and T_{mut} be the total achieved coverage and the elapsed time, for all mutation tasks. Let c_i and t_i be the achieved coverage and elapsed time relating to mutating a seed i. The observed gain of mutating this seed can thus be computed as:

$$g_i^{(ss)} = c_i \cdot \frac{T_{mut}}{C_{mut}} - t_i \tag{6.12}$$

Let $\sigma_{mut}^{(ss)}$ be the standard deviation of the observed gain across all mutation tasks, the final reward of mutating seed i is then computed as:

$$x_i^{(ss)} = \frac{1 - e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}}{1 + e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}} \tag{6.13}$$

Ever-increasing number of arms. Syzkaller starts with no seed in the corpus. The seed corpus is only populated as Syzkaller creates and executes more and more programs. As a result, if we treat the seed selection problem as a Multi-Armed Bandit problem, we would have an ever-increasing number of arms. Although this is not the typical formulation of classic MAB problems, we argue that it is easy to adjust it to fit our needs. Specifically, when a new seed i is added to the seed pool, it starts with a neutral accumulated estimated reward $G_i^{(ss)} = 0$. As a result, its initial weight $w_i^{(ss)}$ will be 1 (in accordance with Algorithm 4). The probability of selecting this seed will initially depend on the accumulated rewards (e.g., $G_j^{(ss)}$) of other seeds already in the corpus. Once seed i is later mutated, the probability of picking seed i will be determined by whether the benefits of attained coverage out-weigh the time cost.

Reset is not necessary in MAB algorithm. Theoretically, given a seed program, the more it is mutated the less likely that future mutations would result in the discovery of new coverage. Therefore, each arm in the seed selection MAB has a diminishing reward. Consequently, there is no point in adopting the *Exp3.1*-style reset mechanism for the seed

selection algorithm (since seeds die out). Our seed selection algorithm simply follows the *Exp3-IX* algorithm, with the only exception being that new arms can be created once a new seed has been added to the corpus.

6.3.4 Implementation

Our implementation of SyzVegas incorporates our reward assessment models and the previously discussed extensions of the *Exp3.1* algorithm on top of Syzkaller. Specifically, we modify Syzkaller based on the version retrieved on 01/08/2020 [9]. Our implementation consists of roughly 1,800 lines of code. Below, we describe some of the subtleties we handled in our implementation.

Standard deviation computation. During normalization, we need to compute the standard deviation of all the previously observed rewards as shown in Equation 6.11 and 6.13. Keeping track of all the reward values is impractical as it's easy for Syzkaller to execute millions of programs. In addition, these numbers need to be synced with the host machine and restored if the fuzzer VM/device crashes or disconnects. Fortunately, we only need to keep track of the 1) total number of observations n, 2) $\sum g$ and 3) $\sum g^2$. We can then compute the standard deviation as:

$$\sigma(g) = \sqrt{E(g^2) - E^2(g)} = \sqrt{\sum g^2/n - (\sum g/n)^2}. \tag{6.14}$$

Outlier Handling. Programs on the fuzzer VM/device can take different amounts of time to execute. In some cases, a program can take several seconds for execution. Although this happens rarely, a mere (insignificant in number) few extreme cases can severely throw off the time estimation and standard deviation, hurting our task selection and seed selection

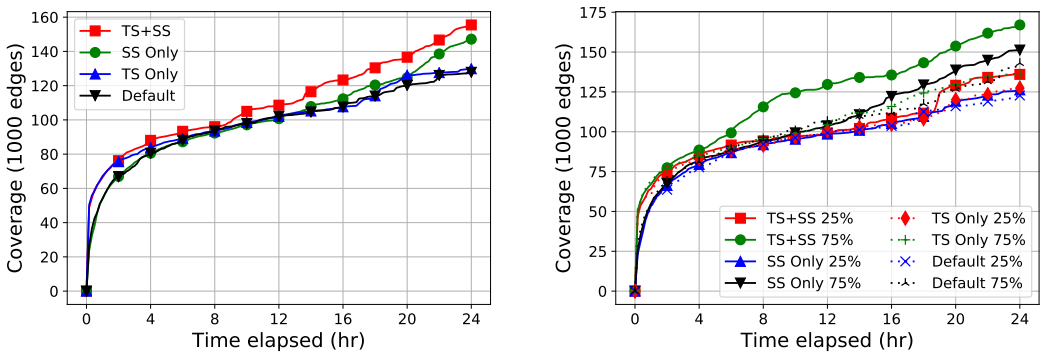
algorithms. Thus, it is crucial that we detect these outliers and prevent them from damaging the integrity and effectiveness of our algorithms.

According to our measurements, triage is the most costly type of task and its execution time can vary greatly. If we use the "3rd quartile + interquartile range" method to detect outliers, we would set the threshold at 0.32 seconds. To allow us some more flexibility without compromising experimental integrity, we set one second as the outlier detection threshold. For any task that costs more than one second, we treat it as if it costs one second. During our experiments, we observe that this cost bounding will only affect less than 1% of all tasks.

6.4 Evaluation

6.4.1 Fuzzing the Linux Kernel for 24 hours

First, we conduct a 24-hour fuzzing experiment on the full Linux kernel to perform a systematic evaluation and analysis of SyzVegas. We run our experiments on several servers. Each server is equipped with an Intel(R) Xeon(R) CPU E5-2680 v4 2.40GHz CPU. We conduct 10 runs of three different setups of SyzVegas and the default Syzkaller, for a total of 40 runs in parallel. In each run, we create one fuzzer VM that uses 2 cores and 2 GB memory. We target the full Linux kernel at version 5.4.8, compiled using the `defconfig` and `kvmconfig` with only the necessary additions [44] to provide coverage information and make Syzkaller functional. For MAB seed selection, we choose an implicit exploration factor $\gamma = 0.05$ and a growth factor $\eta = 0.1$.



(a) Median

(b) 25 and 75 percentile

Figure 6.4: Median, 25/75 percentile of coverage reached for fuzzing Linux kernel for 24hrs. Comparison of SyzVegas with/without task selectiion (TS) and seed selection (SS).

Coverage growth Figure 6.4(a) shows the median coverage growth reached after fuzzing the Linux kernel for 24 hours. Figure 6.4(b) shows the 25 and 75 percentiles instead. From these two figures, we have several interesting observations:

- MAB task selection works best at the early stages of fuzzing. However, the initial advantage is lost as fuzzing reaches its later stages.
- MAB seed selection has little effect in the first few hours. However, as we run for longer, seed selection begins to increase coverage growth, reaching 15.1% at 24 hours for the median.
- Combining MAB task and seed selection produce considerable improvements in coverage growth, reaching around 21.6% at 24 hours for the median. Interestingly, while MAB task selection does not provide any advantage by itself at 24 hours, combining it with MAB seed selection significantly outperforms Syzkaller with MAB seed selection only (no task selection).

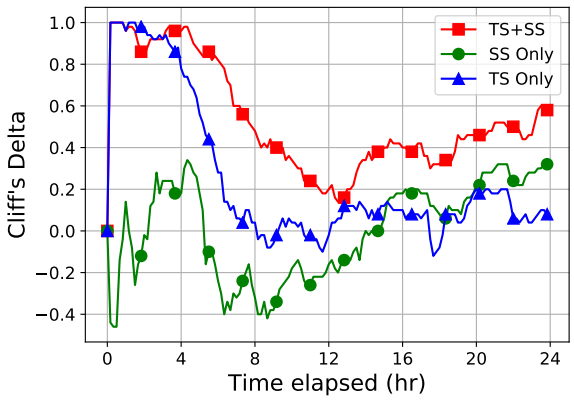
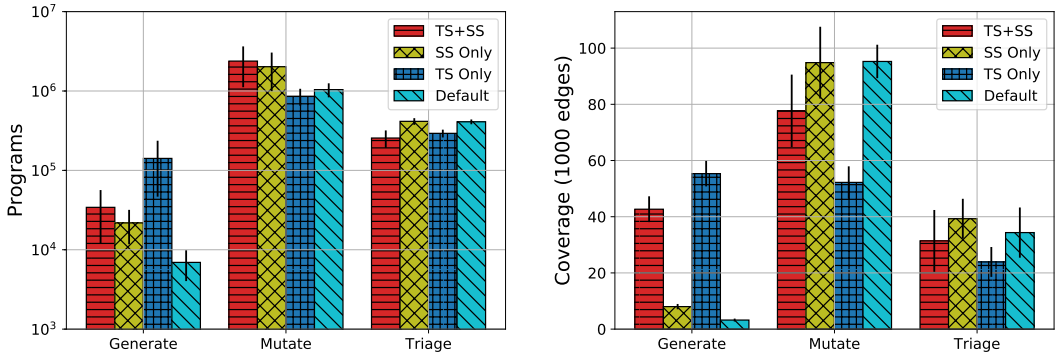


Figure 6.5: Median, 25/75 percentile and Cliff's delta of coverage reached for fuzzing Linux kernel for 24hrs. Comparison of against the default Syzkaller.

Since luck plays a prominent role in the coverage growth of fuzzing, researchers propose that statistical methods be utilized to determine the likelihood of the observed differences in coverage [64]. To evaluate whether the coverage advantage of SyzVegas is consistent across all runs, we compute Cliff's delta [37] between runs with MAB task and/or seed selection against the default Syzkaller. Cliff's delta lies in the range $[-1, 1]$ and represents the pair-wise comparison result between runs (in our case between our setup and runs with default Syzkaller). A higher Cliff's delta means that our setup is more likely to outperform the default Syzkaller. Figure 6.5 demonstrates the *Cliff's Delta* of our setups against the default syzkaller. The Cliff's delta result verifies that our observations in Figure 6.4(a) and 6.4(b) have associated high confidence.

Figure 6.6(a) shows the number of programs executed by different types of tasks. Understandably, all of our optimizations generate more programs by giving a higher priority to generation and/or removing the mandatory smash mutation. An interesting observation is that when seed selection is used, SyzVegas executes more programs in total than the



(a) Programs executed

(b) Coverage reached

Figure 6.6: Statistics of program execution.

default Syzkaller. This is primarily due to seed selection favoring mutating seeds with low execution time, allowing SyzVegas to perform more mutations. This reflects our design goal of optimizing coverage-time efficiency of tasks.

Figure 6.6(b) breaks down the coverage by the task types. Based on our observations, MAB task selection significantly shifts the workload from mutation to generation, giving generation a 20 times boost in terms of the coverage found. This comes with a sacrifice though, in the form of a 50% reduction in coverage discovered by mutations. Fortunately, seed selection compensates for this loss, bringing the power of mutations back to its original level.

Interestingly, we find that when MAB task selection is present, generation produces a huge amount of coverage. However, when we take a look at the number of programs executed (Figure 6.6(a)), SyzVegas still favors mutation. If we break down the coverage achieved by the different tasks over time, as shown in Figure 6.7, we observe that the coverage reached by generation almost exclusively achieved in the first 2 hours when the

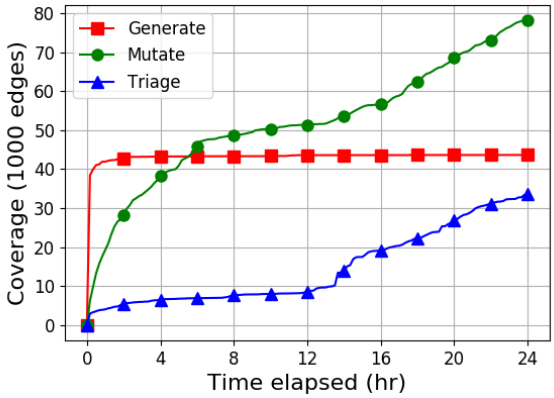


Figure 6.7: Coverage growth by task for SyzVegas with both task selection and seed selection

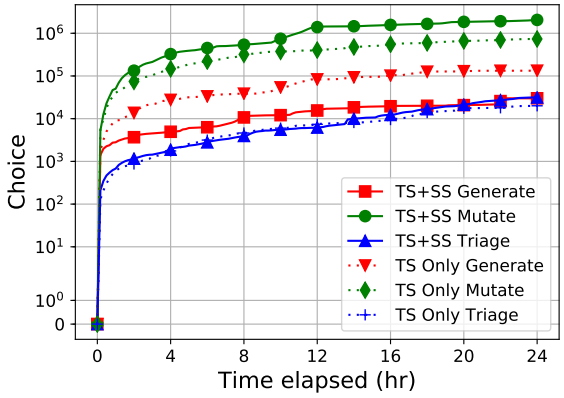
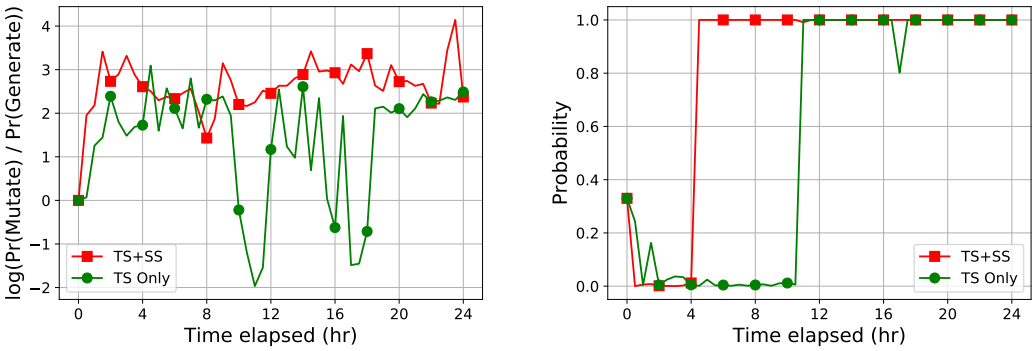


Figure 6.8: MAB task selection choices.

kernel code space is not explored much, and finding new coverage is simple. This, as we will show later, is due to task selection performing plenty of generation at the early stages.

MAB Task selection. We then take a look at the inner workings of MAB task selection. In particular, we want to understand how much is the probability assigned by the task selection algorithm to each type of task.

Figure 6.8 captures the choices made by task selection algorithm. We observe that at the beginning of fuzzing, task selection quickly "pulled" the generation "arm" more than



(a) Generation vs mutation probabilities

(b) Triage probabity

Figure 6.9: Statistics of MAB task selection.

1000 times, giving generation much higher priority than the default Syzkaller. Triage, on the other hand, is less-favored compared with generation at the beginning, but it starts to slowly catch up as fuzzing goes on.

Figure 6.9(a) illustrates how MAB task selection balances generation and mutation over time, with or without seed selection. In the beginning, generation and mutation are initialized to have the same probability. With the help of the associated seed selection, the task selection algorithm quickly determines that mutation is the better option, giving it around a 500 times higher likelihood. Without seed selection, however, the task selection algorithm favors generation much more, even giving it a higher probability of being invoked than mutation occasionally. This is expected due to the issues from the default seed selection algorithm, as discussed in Section 6.2.2. Without the improved seed-selection algorithm, mutations are less effective in finding new coverage and thus fall out of favor.

Figure 6.9(b) shows the probability change over time for triage, averaged across all 10 runs for each setup. Triage is not always available (when no more interesting programs

are in the work queue), and thus Figure 6.9(b) only focuses on its probabilities when it is a valid arm. In the beginning, task selection gives triage a few chances before assigning it a very low priority, favoring generations and mutations, much more. At this stage, generation and the initial seeds (accumulated from the few triage tasks) are still very powerful, causing the task selection algorithm to give generation and the mutation of these initial seeds better rewards than triage.

But as we run out of the power of generation and these initial seeds, generation and mutation begin to receive negative rewards (no new coverage yet but a time cost is incurred). Triage will then be favored naturally. Its ability to generate new seeds and maintain a diverse seed pool becomes essential here, to discovering new coverage. This effect is especially prominent when there is MAB seed selection that makes mutation more effective, while the power of the initial seeds are exhausted faster, causing triage to be invoked earlier on. Thanks to its exponential weight growth feature, SyzVegas quickly adjusts its policy giving triage the absolute priority (when appropriate) just like the default Syzkaller. Note that a near 100% triage probability does not mean SyzVegas will not execute anything else. Triage tasks are created by generation and mutation and are not always available (when no interesting programs are pushed into the work queue). When SyzVegas has no more triage tasks to schedule, it will select generation or mutation tasks.

Further, what is surprising is that according to the task selection algorithm, the power of generation and initial seeds can last as long as 4 to 10 hours, and the default Syzkaller does not tap into this power nearly as much. As Syzkaller keeps evolving with improved templates and mutation strategies, the power of generation and mutation may

change over time as well, making auto-tuning the task selection the best long-term option moving forward (instead of hand-picking a threshold).

Overall, we find that the main effects of MAB task selection are **performing more generations** and **deferring triages** at the very early stages of fuzzing. After a few hours, however, MAB task selection eventually converges to the same policy of the default Syzkaller. Triage takes absolute priority, while mutation tasks are heavily favored over generation tasks. This behavior is the most prominent when combined with seed selection, where mutations are more rewarding.

We now attempt to understand why combining MAB task selection and seed selection significantly outperforms MAB seed selection only, even when MAB task selection is losing its effectiveness and converging toward a policy similar to that of the default Syzkaller. As discussed before, the main effect of task selection is performing more generation tasks and fewer triage tasks at the early stages of fuzzing, which heavily impacts the initial seeds added into the corpus. Therefore, we also hypothesize that these early-stage seeds have long-term benefits, similar to previous researches [83].

Seed power. Mutation, the main workhorse of finding new coverage, requires seed programs to function. Therefore, accounting for the "power" of seeds, i.e., how much coverage a seed can produce through mutation, has a significant influence on fuzzing efficiency.

Figure 6.10 shows the number of seeds generated by the fuzzer throughout the 24-hour. We find that with MAB task selection, Syzkaller produces much fewer seeds. Figure 6.11(a) illustrates the distribution of new coverage attained by mutating these seeds, a.k.a. seed power. As expected, the MAB seed selection algorithm improves seed power by

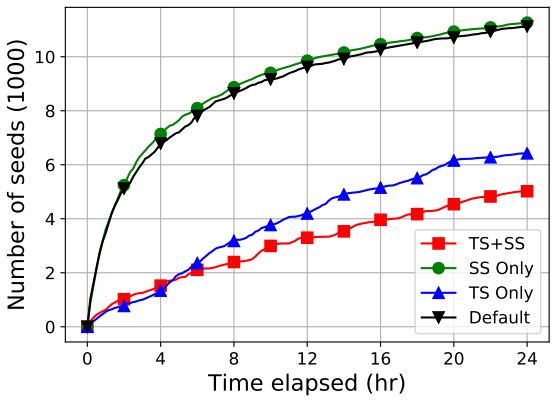
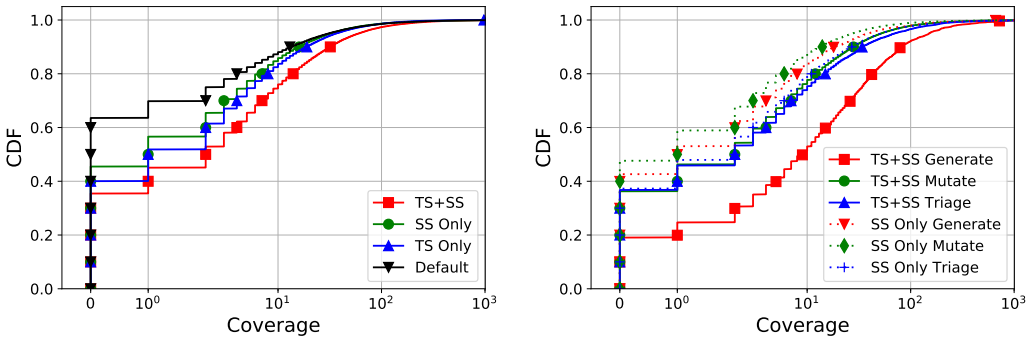


Figure 6.10: Seed number growth over time.



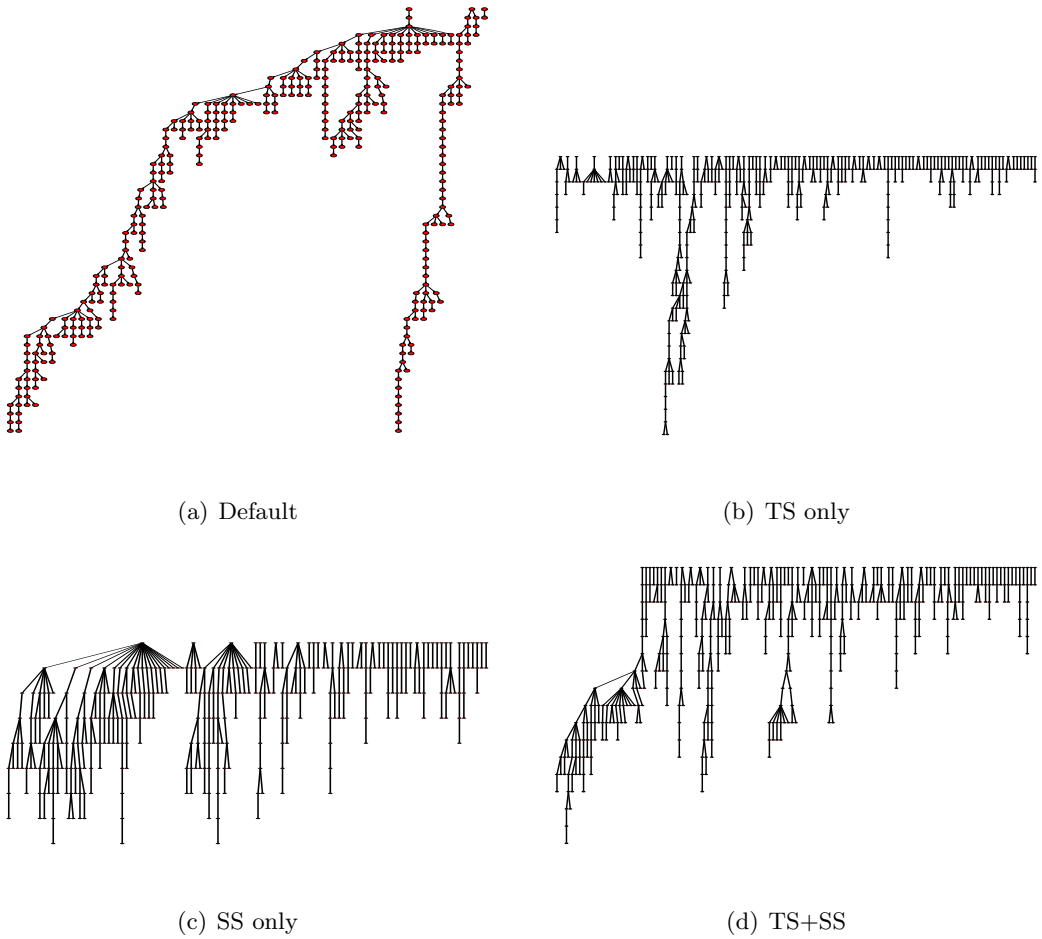
(a) Seed power comparison.

(b) Breakdown: TS + SS vs SS only

Figure 6.11: Coverage gained (seed power) by mutating seed programs.

preferring good seeds for mutation. Interestingly, we find that adding MAB task selection contributes to improving the seed power, despite not directly affecting seed selection. In other words, the coverage benefits induced by MAB task selection must come from its contribution to seed quality; this is where the initial generations performed by MAB task selection help, by creating some very powerful seeds.

We break down the seed power distribution (how much new coverage a seed yields) based on the origin of the seeds in Figures 6.11(b). Clearly, we observe that MAB task



(a) Default (b) TS only

(c) SS only (d) TS+SS

Figure 6.12: Evolution forests down-sampled to around 500 nodes.

selection brings an increase in the power of the seeds originating from generation. This verifies our hypothesis as well as our motivation, where more early generations is beneficial to the Syzkaller fuzzing process.

Program evolution. To further understand the impact of MAB task selection and seed selection, we take a look into the program evolution in Syzkaller. In Syzkaller, everything starts from generated programs, who then go through a series of minimizations and mutations, creating a tree-like structure. Multiple generations will result in a program evolution

forest consisting of multiple trees. We pick one run for each setup and construct the program evolution forest, down-sample it to around 500 nodes and display them in Figure 6.12.

We observe that different strategies have very different approaches to program evolution. The default Syzkaller (Figure 6.12(a)), as discussed earlier in Figure 6.2.2, favors a depth-first approach. It performs very few generations (13 trees before sampling) and is quite biased when it comes to exploration (a.k.a. mutation). With only task selection (Figure 6.12(b)), SyzVegas performs the most generation tasks and created the biggest number of trees (789 before sampling), but spend less time exploring them while suffering from the same biased exploration of the default Syzkaller. With only seed selection (Figure 6.12(c)), a reasonable number of trees (202 before sampling) are created while trees are more balanced. However, it is clear that the tree created by the very first generation is explored much more in-depth than the latter trees. Finally, with both task and seed selection, SyzVegas combines both the large tree numbers because of scheduling, and the exploration balance from seed selection. Specifically, with more generations at the beginning, SyzVegas is able to turn more of these generations (347 before sampling) into program evolution trees.

Performance overhead. Finally, we evaluate the overheads of the MAB task selection and the seed selection algorithms. The overheads come from two sources: 1) computing and updating weights and probabilities for tasks and seeds and 2) synchronizing the MAB status between the fuzzer VM and the manager host. The syncing overhead is closely related to how often the fuzzer crashes, as when does, it needs to fetch all information about the seed corpus from manager host all over again. During the 24 hour experiment, updating costs

around 9 minutes while synchronizing costs 22 minutes. Overall, the overhead of MAB task selection and seed selection is less than 2.1%.

When it comes to memory, SyzVegas needs to store some additional information such as the weights of the arms, the total reward thus far, etc.. Copies of these records must be maintained by each fuzzer VM and the manager host, in case the fuzzer crashes. For task selection, we use a constant 250 bytes to store the necessary information. For seed selection, we use 104 bytes of additional memory for each seed. With ∼5,000 seeds created by SyzVegas in 24 hours, we incur ∼520 KB of memory overhead per VM/manager.

6.4.2 Fuzzing Linux Kernel for 5 days

To further study the long-term performance of SyzVegas, we run a 5-day fuzzing experiment on the full Linux 5.4.8 kernel. We run this specific experiment on a different server, equipped with Intel(R) Xeon(R) Gold 6248 2.50GHz CPU. We conducted 10 runs of both SyzVegas and the default Syzkaller, for a total of 20 runs in parallel. In each run, we created one fuzzer VM that uses 2 cores and 2 GB memory. For MAB seed selection, we use the same implicit exploration factor $\gamma = 0.05$ and a growth factor $\eta = 0.1$ as before.

Figure 6.13 shows the median coverage growth. Over the course of fuzzing, SyzVegas produces 35,000 (16.1%) more branch coverage, in the median case. Table 6.2 lists the unique crashes we find. SyzVegas discovers 6 unique crashes, which the default Syzkaller failed to find. Sadly, through automated reproduction and manual verification, we have so far only been able to successfully verify one unfixed bug. Many of them cannot be reproduced by Syzkaller after the OS is reset. This is a known deficiency of Syzkaller, as

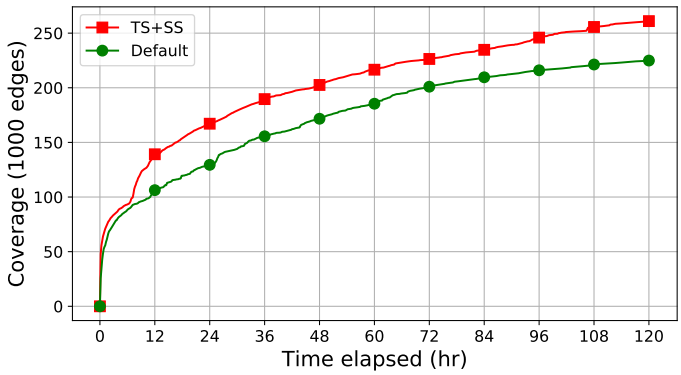


Figure 6.13: Median coverage reached for fuzzing Linux kernel for 5 days. Comparison of SyzVegas with default Syzkaller

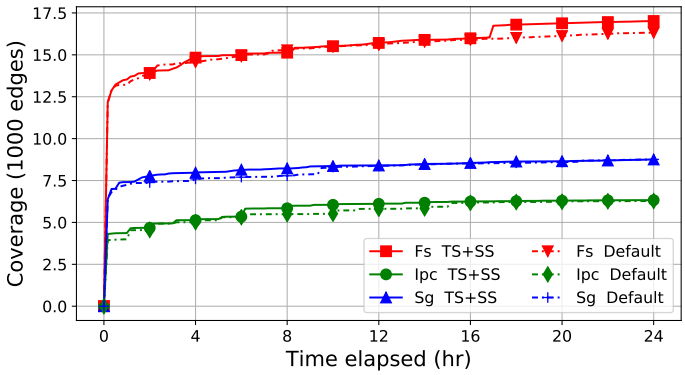


Figure 6.14: Median coverage reached for fuzzing Linux sub-systems for 24 hours. 5 runs for each setup.

`syzbot` [45], the official automated fuzzing and reporting tool, also reported 6 unreproducible crashes we discovered.

6.4.3 Fuzzing Linux Kernel Modules

We also compared the fuzzing effectiveness of SyzVegas and the default Syzkaller on several Linux kernel modules and sub-systems. We run our experiments on servers with

Table 6.2: Crashes discovered fuzzing Linux kernel for 7 days

Crash	Function	Discovered		Status
		SyzVegas	Default	
Register deref	interrupt_entry	Y	N	NR
UAF Read	relay_switch_subbuf	N	Y	NR*
UAF Read	ata_scsi_mode_select_xlat	Y	N	Not fixed*
UAF Read	__locks_wake_up_blocks	N	Y	NR*
slab-out-of-bounds	mpol_parse_str	Y	Y	Not fixed*
slab-out-of-bounds	mpol_to_str	Y	Y	Not fixed
Protection fault	unlink_anon_vmas	Y	N	NR
Warning	perf_group_attach	Y	Y	NR*
Warning	generic_make_request_checks	Y	Y	NR*
Warning	restore_regulatory_settings	Y	N	NR*
Warning	xfrm_policy_insert_list	Y	N	NR*
RCU stall	snd_seq_write	Y	N	Pending*

*: Reported by syz-bot. NR: Not-reproducible. Pending: Reproducible but pending analysis.

Intel(R) Xeon(R) CPU E5-2680 v4 2.40GHz CPU. We conduct 5 runs of SyzVegas and the default Syzkaller, for a total of 10 runs in parallel for each module/sub-system.

Figure 6.14 shows the coverage growth of fuzzing the file system, the IPC module, and the SCSI generic (sg) driver. We observe that SyzVegas performs slightly better in terms of coverage, while both SyzVegas and default Syakaller are having trouble finding new coverage after a few hours. Table 6.3 shows the crashes we found using both SyzVegas and the default Syzkaller. Despite not being able to drastically improve coverage, SyzVegas does find 5 crashes not detected default Syzkaller. Among them, we have so far verified one unfixed bug. Interestingly, the bug in `ata_scsi_mode_select_xlat()` is found

Table 6.3: Crashes discovered fuzzing Linux sub-systems for 24 hours

Crash	Function	Discovered		Status
		SyzVegas	Default	
NULL pointer	ip_rcv_finish	Y	N	Not fixed
Kernel paging	sel_netnode_find	Y	N	NR
Protection fault	d_hash_and_lookup	Y	N	NR
Protection fault	pid_vnr	Y	Y	Not fixed
Protection fault	__fget_light	Y	N	NR
RCU stall	sys_umount	Y	Y	Pending
RCU stall	__do_sys_newstat	Y	N	Pending
UAF Read	ata_scsi_mode_select_xlat	Y	Y	Not fixed*

*: Reported by syz-bot. NR: Not-reproducible. Pending: Reproducible but pending analysis.

by the default Syzkaller only during the more focused kernel module fuzzing, while SyzVegas is able to discover it in both module fuzzing and full kernel fuzzing. This is another example of the benefits of the breadth-first approach of SyzVegas.

6.5 Related Work

Applying reinforcement learning to fuzzing. Researchers have attempted to apply reinforcement learning techniques to perform seed selection. Woo et al. [103] use MAB algorithm to perform crash-based seed selection for black-box fuzzing within a fixed run/time budget. In our kernel-fuzzing context, however, the ability to find crashes is a sub-optimal measurement to seed power since 1) the kernel codebase is huge and crashes are hard to find and 2) assigning more probability on the seeds finding crashes may only trigger the

same crashes repeatedly. Patil et al. [84] treat the problem of assigning rewards the fuzzing iterations to a test case as a "Contextual Bandit" problem. However, their reward assignment does not take the time cost into consideration. Comparing to previous researches, SyzVegas focuses on optimizing the branch coverage growth during fuzzing process by an intuitive reward assessment model capable of balancing the coverage achieved and time cost. We also addressed the task selection problem unique to Syzkaller, including the problem of reward distribution between mutation and triage.

Böttinger et al.,[31] use Q-learning [102] to learn a policy for choosing mutation operators. Karamcheti et al. [61] apply what is called a Thompson Sampling, bandit-based optimization approach to fine-tune the mutation operator distribution. MOpt [74] utilized a customized Particle Swarm Optimization (PSO) algorithm to determine the optimal distribution of mutation operators. SyzVegas focuses on scheduling the right type of work and choosing the right seed for mutation, independent of the distribution of mutation operators. In theory, SyzVegas can work perfectly alongside any algorithm aiming to optimize the mutation operator distribution.

Seed selection optimizations. Researchers have also focused on optimizing different mutation strategies. Rebert et al. [87] explored several different seed selection algorithms and measured their qualities using linear programming. DigFuzz [110] uses a Monte-Carlo-based algorithm to prioritize favorable paths, and determine the seeds that are more valuable for future mutations. AFLFast [30] treats fuzzing as a Markov chain problem, modeling the probability that fuzzing the seed that exercises one path, generates an input that exercises another path. SyzVegas treats the inner workings of the seeds as a black box and model

the coverage achieved and the time cost of mutating different seeds. It does not require any program analysis technique, and can easily adapt to different fuzzing targets.

Kernel fuzzing. Syzkaller provides a great foundation for coverage-based kernel fuzzing. There are several works built upon it, to improve kernel fuzzing from different aspects. FastSyzkaller [69] combines Syzkaller with an N-Gram model, to optimize the test case generation process. Moonshine [83] tries to improve the quality of the initial seeds in Syzkaller by "distilling" seeds from system call traces of real-world programs. RAZZER [60] combines fuzzer and static analysis, to detect race bugs in kernel. Difuze [38] utilizes static analysis to compose correctly-structured inputs in the userspace, to explore kernel drivers. SyzVegas focuses on automating critical fuzzing decisions in Syzkaller, and demonstrates its short-term and long-term effects on coverage growth, and targets improvements of the same.

Besides Syzkaller, there exist other fuzzers for OS kernels. Trinity [24], iknowthis [19], KernelFuzzer [21], and sysfuzz [23] are built with hard-coded rules and grammars. kAFL [88], TriforceLinuxSyscallFuzzer [47], TriforceAFL [57], on the other hand, are based on or inspired by AFL. Given that Syzkaller is the state-of-the-art, we decide to implement our reinforcement learning based solution on top of it. However, we believe the idea generalizes to these other kernel fuzzers as they likewise encode numerous decisions and parameters (e.g., which syscall to insert and when to mutate the value of an argument).

6.6 Conclusions

In this paper, motivated by observations that kernel fuzzing strategies have numerous fixed decisions and hard-coded parameters, we propose SyzVegas to dynamically choose the right fuzzing task in conjunction with the right seed, in Syzkaller. Towards this, we model the specific fuzzing tasks as a multi-armed-bandit problem, which allows the system to learn the effective strategies and adapt over time, using a novel, yet intuitive reward assessment model to capture benefits and costs. We evaluate SyzVegas on Linux kernel version 5.4.8 and several of its modules. We demonstrate that SyzVegas has a low 2.1% performance overhead and makes considerable improvements in terms of coverage achieved and crashes found, relative to the default Syzkaller. We believe our improved coverage growth will be very beneficial for researchers and developers for large-scale, continuous fuzzing projects.

Algorithm 3 Task selection Algorithm

for all $r = 1, 2, ...$ **do**

  $\hat{G}_{gen}(0), \hat{G}_{mut}(0), \hat{G}_{tri}(0) \leftarrow 0$

  $t \leftarrow 0$

  $\gamma \leftarrow 2^{-r}$

  $\eta \leftarrow 2 \times \gamma$

  $\hat{G}_{threshold} \leftarrow \frac{3\,ln3}{e-1} \cdot 4^r - \frac{1}{3\,\gamma}$

  while $max_i(|\hat{G}_i|) < \hat{G}_{threshold}$ **do**

    $w_i(t) \leftarrow e^{\eta \hat{G}_i(t)}$

    $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}$

    Draw i_t according to $pr_{gen}(t), pr_{mut}(t), pr_{tri}(t)$

    if $i_t = gen$ **then**

      Receive reward $x_{gen}(t)$

      $\hat{G}_{gen}(t+1) \leftarrow \hat{G}_{gen}(t) + x_{gen}(t)/(pr_{gen} + \gamma)$

    else if $i_t = tri$ **then**

      Receive initial reward for triage $x_{tri}(s)$

      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{tri} + \gamma)$

    else if $i_t = mut$, Seed s is selected **then**

      Receive reward detlas $x_{mut}(t), x_{tri}(t)$

      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{mut} + \gamma)$

      $\hat{G}_{mut}(t+1) \leftarrow \hat{G}_{mut}(t) + x_{mut}(t)/(pr_{mut} + \gamma)$

    end if

    $t \leftarrow t + 1$

  end while

end for

Algorithm 4 Seed Selection Algorithm

Require: $\gamma, \eta \in (0, 1)$

1: **for all** $t = 1, 1, 2$ **do**

2: $w_i \leftarrow e^{\eta \hat{G}_i}$

3: $pr_i \leftarrow \frac{w_i}{\sum_j w_j}$

4: Draw seed i_t randomly according to pr_i

5: Receive reward $x_i(t)$

6: $\hat{G}_i(t+1) \leftarrow \hat{G}_i(t) + \frac{x_i(t)}{pr_i + \gamma}$

7: **end for**

Chapter 7

Conclusion

In this thesis, we explored the possible attacks and applications of existing CPU side-channels and conducted several systematic studies.

First, we propose a concept of attacking Android graphic buffer using prime+probe attack. We identify some key challenges of implementing an end-to-end attack, including the requirement of CFS scheduling slowdown as well as the ability to monitor the entire CPU LLC in the presence of prefetching.

We thus propose PAPP: a prefetcher-aware prime and probe cache side-channel attack. PAPP performs systematic profiling of CPU cache prefetcher and replacement policy. We show that PAPP is able to construct a prime+probe strategy that is resistant to the interference of prefetcher. We evaluate PAPP on real-world systems using cache side-channel vulnerability (CSV) metric and demonstrates that PAPP doubles the information leakage comparing to traditional prime and probe implementations. We hope that PAPP can be used in new attacks and applications to provide efficient cache monitoring.

We then discover a previously unknown type of potent side-channel that allows an attacker to use the flush+reload attack to perform cross-process timing measurements on sensitive functionalities inside shared graphic libraries. The attack facilitates the inference of a user's keystrokes when the typed keys are rendered on the screen or going through graphic pre-processing. The attack hinges on utilizing machine learning techniques to discover execution-time based side-channels inside graphic libraries. We have completely automated the discovery of such side-channels and even the generation of exploits. We validate that our attack is viable on real-world applications on multiple platforms and demonstrate its high accuracy in predicting user input in practice. This attack could potentially affect a large user population of the considered applications.

Next, we propose to utilize CPU side-channels as feedback sources when fuzzing un-modifiable binaries. We identify several possible scenarios where this technique might be useful and state the technical challenges the need to be addressed. We also perform an initial evaluation on real-world systems and make a surprising discovery on the opportunities in improving kernel fuzzing efficiency.

Finally, we propose SyzVegas to dynamically choose the right fuzzing task in conjunction with the right seed, in Syzkaller, the state-of-art kernel fuzzer. Towards this, we model the specific fuzzing tasks as a multi-armed-bandit problem, which allows the system to learn the effective strategies and adapt over time, using a novel, yet intuitive reward assessment model to capture benefits and costs. We evaluate SyzVegas on the Linux kernel and several of its modules and demonstrate that SyzVegas has a low performance overhead and makes considerable improvements in terms of coverage achieved and crashes

found, relative to the default Syzkaller. We believe our improved coverage growth will be very beneficial for researchers and developers for large-scale, continuous fuzzing projects.

Ultimately, we demonstrate that CPU side-channels have plenty of potential and can be utilized in interesting attacks and applications. We also learned that a successful end-to-end attack using CPU side-channel is not a trivial task and require many detailed technical challenges being addressed. We hope our work could inspire more studies of novel attacks and applications using CPU side-channels.

Bibliography

[1] Arm cortex-a57 mpcore processor technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488f/DDI0488F_cortex_a57_mpcore_trm.pdf.

[2] Citic bank app. https://shouji.baidu.com/software/11576280.html.

[3] Demo videos of attack. https://sites.google.com/view/swtwmyc/home.

[4] Github repository. https://github.com/CacheAttackGraphics/KeystrokeAttack.

[5] Glyph hell: An introduction to glyphs, as used and defined in the freetype engine. http://chanae.walon.org/pub/ttf/ttf_glyphs.htm.

[6] Intel® 64 and ia-32 architectures optimization reference manual). https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Most common passwords list. http://www.passwordrandom.com/most-popular-passwords/page/1.

[8] Reliance global call. https://www.relianceglobalcall.com/.

[9] Syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[10] Technologies — big.little – arm developer. https://developer.arm.com/technologies/big-little.

[11] Wayland. https://wayland.freedesktop.org/.

[12] X window system. https://en.wikipedia.org/wiki/X_Window_System.

[13] Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2012.

[14] Reliance global call. https://www.relianceglobalcall.com/features/free-international-app-to-app-calls, 2012.

[15] 10 ways to stay safe from cyber attacks. https://medium.com/all-technology-feeds/10-ways-to-stay-safe-from-cyber-attacks-6449219ceb54, 2017.

[16] Honor/huawei's bootloader unlock page disappears with no explanation. https://www.xda-developers.com/honor-huawei-bootloader-unlock-page-disappears, 2018.

[17] Huawei clarifies it has "no plans" to unlock the mate 30 series' bootloader. https://www.androidauthority.com/huawei-mate-30-pro-bootloader-1031805, 2019.

[18] American fuzzy loop. http://lcamtuf.coredump.cx/afl, Online.

[19] iknowthis linux systemcall fuzzer. https://github.com/rgbkrk/iknowthis, Online.

[20] Kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html, Online.

[21] Kernel fuzzer: Cross platform kernel fuzzer framework. https://github.com/FSecureLABS/KernelFuzzer, Online.

[22] libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, Online.

[23] sysfuzz: A prototype system call fuzzer. https://github.com/markjdb/sysfuzz, Online.

[24] Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, Online.

[25] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *Proc. ISPASS*, 2014.

[26] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *IEEE Symposium on Security and Privacy*, 2004.

[27] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 322–331. IEEE, 1995.

[28] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The non-stochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[29] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2014.

[30] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[31] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018.

[32] Leo Breiman. Random forests. *Machine learning*, 2001.

[33] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009.

[34] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, 2003.

[35] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 2011.

[36] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[37] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[38] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[39] D. Evtyushkin, P. Ponomarev, and N. Abu-Ghazaleh. Jump-over-aslr: Attacking the branch predictor to bypass aslr. In *Proc. Micro*, 2016.

[40] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017.

[41] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156. Citeseer, 1996.

[42] A. Fuchs and R. B. Lee. Disruptive prefetching: Impact on side-channel attacks and cache designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015.

[43] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[44] Google. Setup: Ubuntu host, qemu vm, x86-64 kernel. `https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md`, Online.

[45] Google. Syzbot. `https://syzkaller.appspot.com`, Online.

[46] 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 *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[47] NCC Group. Triforcelinuxsyscallfuzzer. `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer`, Online.

[48] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016.

[49] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[50] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cham, 2016. Springer International Publishing.

[51] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, 2015.

[52] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, Washington, DC, USA, 2011.

[53] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, 2011.

[54] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on aes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2015.

[55] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 2003.

[56] Tzipora Halevi and Nitesh Saxena. A closer look at keyboard acoustic emanations: random passwords, typing styles and decoding techniques. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[57] Jesse Hertz and Tim Newsham. Project triforce: Run afl on everything! `https://github.com/nccgroup/TriforceAFL`, Online.

[58] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[59] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries. *CoRR*, 2017.

[60] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[61] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[62] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016.

[63] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[64] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[65] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. 2019.

[66] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[67] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, 1995.

[68] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[69] Dan Li and Hua Chen. FastSyzkaller: Improving fuzz efficiency for linux kernel fuzzing. *Journal of Physics: Conference Series*, 1176:022013, mar 2019.

[70] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016.

[71] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and . Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018.

[72] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[73] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.

[74] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[75] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[76] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proc. RAID*, 2015.

[77] Gergely Neu. Explore no more: Improved high-probability regret bounds for non-stochastic bandits. In *Advances in Neural Information Processing Systems*, pages 3168–3176, 2015.

[78] Y. Oren, V. P Kemerlis, S. Sethumadhavan, and A. D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[79] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox - practical cache attacks in javascript. *CoRR*, 2015.

[80] Arne Osvik, Dag, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*. Springer, 2006.

[81] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, Berlin, Heidelberg, 2006.

[82] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, 2012.

[83] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[84] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *CoRR*, abs/1806.03806, 2018.

[85] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 2011.

[86] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 543–553, New York, NY, USA, 2016. ACM.

[87] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 861–875, 2014.

[88] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 167–182, 2017.

[89] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating keystroke timing side-channel attacks. In *NDSS*, 2018.

[90] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[91] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, 2018.

[92] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, 2001.

[93] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*. The MIT Press, 2018.

[94] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 2010.

[95] S. P Vanderwiel and D. J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2), 2000.

[96] Pierre Francois Verhulst. Logistic function. `https://en.wikipedia.org/wiki/Logistic_function`, 1838.

[97] Martin Vuagnoux and Sylvain Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[98] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[99] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V. Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2019.

[100] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[101] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[102] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[103] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[104] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy*.

[105] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[106] T. Zhang, F. Liu, S. Chen, and R. Lee. Side channel vulnerability metrics: The promise and the pitfalls. In *Proc. HASP*, 2013.

[107] 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 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[108] Yinqian Zhang, Ari Juels, K Reiter, Michael, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[109] 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 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[110] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[111] Datong P. Zhou and Claire J. Tomlin. Budget-constrained multi-armed bandits with multiple plays. *CoRR*, abs/1711.05928, 2017.

[112] Li Zhuang, Feng Zhou, and J Doug Tygar. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)*, 2009.