SECURING SOFTWARE SYSTEMS BY PREVENTING INFORMATION LEAKS

A Thesis
Presented to
The Academic Faculty

by

Kangjie Lu

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

Georgia Institute of Technology
August 2017

SECURING SOFTWARE SYSTEMS BY PREVENTING INFORMATION LEAKS

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

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

Professor Michael Backes
Center for IT-Security, Privacy and Accountability
Saarland University and MPI-SWS

Professor Debin Gao
School of Information Systems
Singapore Management Univertsity

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

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

Date Approved: 19 July 2017

To my family,

for all the support;

To Wenqing,

who brightened up my Ph.D. life.

ACKNOWLEDGEMENTS

Foremost, I would like to express my deepest appreciation to my admirable advisor, Professor Wenke Lee, who taught me how to conduct high-quality research. Wenke has trained my abilities to find important research problems and reason about the value of research with critical thinking, which are essences for me to be an independent researcher. His words are always enlightening and thought-provoking. He has been supportive to the development of my career and research interests, and generous in providing me research resources. I am very fortunate to have had Wenke as my advisor. I am extremely grateful to my co-advisor, Professor Taesoo Kim, who propelled my research towards success. His solid background and immense knowledge in system security, and his way to approach problems—getting into the root of the matter—have always been inspiring to my research.

I would like to express my special thanks to Professor Debin Gao who brought me to academia and introduced me to system security research. Working with him was delightful and motivating. Many thanks to Professor Michael Backes for always providing constructive comments for our joint, fruitful research. I very much appreciate Professors Mustaque Ahamad and William R. Harris for taking time to serve on my thesis committee. Their insightful feedback has helped me significantly improve this thesis.

The completion of my thesis would not have been possible without the help of my collaborators. I would like to extend my sincere thanks to following brilliant researchers: Professor Chengyu Song, Dr. Zhichun Li, Dr. Stefan Nürnberger, Professor Byoungyoung Lee, Dr. Simon P. Chung, Dr. Tielei Wang, Professor Long Lu, Meng Xu, Marie-Therese Walter, David Pfaff, Professor Vasileios Kemerlis, Professor Xusheng Xiao, Dr. Zhenyu Wu, Professor Zhiyun Qian, Cong Zheng, Dr. Guofei Jiang, Dr. Haoyu Ma, Jianjun Huang, and Professor Xiangyu Zhang.

I cannot leave Georgia Tech without mentioning my lovely lab-mates who have been helpful to my research. I would like to take this opportunity to acknowledge: Wei Meng, Yeongjin Jang, Dr. Changwoo Min, Dr. Sangho Lee, Ming-Wei Shih, Professor Xingyu Xing, Kyuhong Park, Yang Ji, Ruian Duan, Chenxiong Qian, Insu Yun, Ren Ding, Dr. Hong Hu, Wen Xu, Jinho Jung, and Ashish Bijlani.

Last, but not least, I am very much indebted to Wenqing Lei who has always encouraged me to overcome challenges. I will forever be thankful to my parents and sisters in China for their unconditional support and encouragement to pursue my interests.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

For efficiency and flexibility purposes, fundational software systems such as operating systems and web servers are implemented in unsafe programming languages, and system designers often prioritize performance over security. As a result, these systems inherently suffer from a variety of vulnerabilities and insecure designs that have been exploited by adversaries to launch critical system attacks. System attacks constitute a major threat to our cyber world; their two typical goals are to leak sensitive data in victim systems (i.e., data leaks) and to control victim systems (i.e., control attacks).

This thesis aims at defeating both data leaks and control attacks by preventing information leaks. We identified that, in modern systems, preventing information leaks is a general approach to defeating both attacks. Since modern systems widely deploy randomization-based defense mechanisms such as Address Space Layout Randomization (ASLR) [145] and StackGuard [44], leaking a randomized value to bypass these defense mechanisms has become a prerequisite for control attacks. Therefore, preventing information leaks can defeat not only data leaks but also control attacks.

To prevent information leaks, we empirically analyzed the root causes of reported information leaks in software systems and then proposed three ways: (1) eliminating information-leak vulnerabilities, (2) retrofitting system designs against information leaks, and (3) protecting sensitive data from information leaks. While the first two ways aim to fix root causes of information leaks, the third way acts as the second line of defense against information leaks. For each way, we have developed multiple tools. Specifically, we first developed two techniques to eliminate the most common information-leak vulnerabilities in OS kernels: UNISAN [117] eliminates information leaks caused by uninitialized-data

reads, and PointSan [121] eliminates uninitialized-pointer uses. We then developed RuntimeASLR [116] to prevent information leaks caused by the insecure process forking design in daemon servers. At last, we developed ASLR-Guard [118] to stop code-pointer leaks and thus defeat code-reuse attacks, and Buddy [122] to generally detect memory disclosures with replicated execution. While automatically and reliably securing complex systems, all these tools introduce negligible performance overhead.

CHAPTER I

INTRODUCTION

1.1 Problem Statement

Our ubiquitous computer devices are actuated by software systems such as operating systems (OS) and web servers. Since these systems are a foundation of computer devices, their efficiency is particularly important, and they often require direct accesses to hardware resources such as memory. As such, system designers often prioritize performance and flexibility over security, and fundational software systems are implemented in unsafe programming languages (e.g., C/C++). As a result, these systems inherently suffer from a variety of insecure designs and vulnerabilities. For example, according to the U.S. National Vulnerability Database, more than 1,800 vulnerabilities in the Linux kernel alone have been reported. System attacks exploiting insecure designs and vulnerabilities have become a critical threat. The past several years have continuously witnessed critical attacks targeting systems belonging to individuals, enterprises, and even government agencies. A single system attack may cause billions of loss. In general, system attacks have two typical goals: to control victim systems (i.e., control attacks) and to leak the sensitive data in victim systems (i.e., data leaks). It is pressing to protect our systems from data leaks and control attacks.

1.2 Thesis Overview

This thesis work aims to defeat both data leaks and control attacks by preventing information leaks. Since in modern software systems, randomization-based defense mechanisms such as Address Space Layout Randomization (ASLR) [145] and StackGuard [44] are widely deployed, leaking a randomized value to bypass these defense mechanisms has become a

Table 1: A summary of causes of reported information leaks in computer devices.

Category	Type	Example	Defense/mitigation example
Vulnerabilities (implementation errors)	Memory error	Uninitialized-data read	UNISAN [117], POINTSAN [121]
	Logic error	Missing check	Model checking techniques
	Hardware error	Row hammer [93]	G-CATT [24]
Design flaws	Specification issue	Uninitialized padding	UNISAN [117]
	Organization issue	Refork-on-crash [116]	RUNTIMEASLR [116]
	Mechanism issue	Deduplication+COW [22]	HexPADS [147]
Side channels	Cache	AnC [73]	CacheBar [195]
	Timing	Loop timing [157]	ASLR-GUARD [118]
	Differential fault	Crashing [20]	RUNTIMEASLR [116]

prerequisite for control attacks. By preventing information leaks, we ensure that the prerequisite is unsatisfiable to control attacks, and thus we can defeat them. More importantly, unlike traditional control attack defense mechanisms such as control-flow integrity [2] that typically require extensive runtime checks, information-leak prevention can be achieved in more efficient ways. It is intuitive that preventing information leaks also defeats data leaks. Therefore, preventing information leaks can be a general and efficient approach to defeating both data leaks and control attacks.

To prevent information leaks, we first empirically analyzed the causes of reported information leaks in systems, and summarize them in Table 1. The main causes include information-leak vulnerabilities (i.e., implementation errors), design flaws, and side channels. Based on our cause analysis of information leaks, we propose to prevent information leaks in three ways: (1) eliminating information-leak vulnerabilities, (2) retrofitting system designs against information leaks, and (3) protecting sensitive data from information leaks. The first two ways aim at fixing root causes of information leaks—vulnerabilities and insecure designs in system code. Since completely fixing some information leaks such as the ones caused by side channels is hard, the third way tries to protect certain sensitive data even in the presence of information leaks, which acts as the second line of defense.

The following three sections introduce our research approaches for each way of preventing information leaks. We have developed multiple practical systems, which automatically

prevent information leaks in complex systems while preserving their reliability and efficiency.

1.2.1 Eliminating Information-Leak Vulnerabilities

One root cause of information leaks is that system code has information-leak vulnerabilities. Attackers can exploit these vulnerabilities to leak sensitive data or to leak randomized value to break defense mechanisms. Hence, an intuitive way to prevent information leaks is to eliminate information-leak vulnerabilities, and thereby we propose two techniques, UNISAN and POINTSAN, to eliminate the most common information-leak vulnerabilities in OS kernels. While UNISAN completely eliminates information leaks caused by uninitialized data reads, POINTSAN eliminates the ones caused by uninitialized pointer dereferencing in a practical manner.

1.2.1.1 UNISAN*: Eliminating Uninitialized Data Leaks*

OS kernels are the de facto trusted computing base for most computer systems, which contain lots of sensitive data such as cryptographic keys and file caches. On the other hand, OS kernels enforce defense mechanisms such as ASLR and StackGuard to mitigate various attacks (e.g., code-reuse attack and privilege-escalation attack). The effectiveness of these mechanisms relies on the confidentiality of the randomized pointers and canaries. Unfortunately, OS kernels have many information-leak vulnerabilities. In practical, sensitive data in OS kernels can be leaked, and defense mechanisms are often broken because the randomized pointers and canaries can be leaked. Therefore, it is important to eliminate information-leak vulnerabilities in OS kernels.

Our study reveals that most (about 60%) kernel information leaks are caused by uninitialized data reads—kernel space is shared by all processes; reading an uninitialized variable may read data of other processes. We propose UNISAN, a novel, compiler-based approach to eliminating all information leaks caused by uninitialized reads in OS kernels. UNISAN achieves this goal by using byte-level, flow-sensitive, context-sensitive, and field-sensitive

initialization analysis and reachability analysis that check whether an allocated object has been fully initialized when it leaves kernel space; if not, UNISAN automatically instruments the kernel code to zero-initialize the object when it is allocated. UNISAN's analyses are conservative to avoid false negatives and are robust by preserving the semantics of OS kernels. Since UNISAN zero-initializes only a small portion of allocations, it imposes negligible performance overhead to the secured kernels.

1.2.1.2 POINTSAN*: Eliminating Uninitialized Pointer Uses*

A common type of memory error in OS kernels is using uninitialized pointers (uninitialized use). Uninitialized uses not only cause undefined behaviors but also impose a severe security risk if an attacker takes control of the uninitialized pointers—an attacker can exploit an uninitialized pointer use to achieve arbitrary read (i.e., information leak), write, and execution. Our study shows that uninitialized pointers can be reliably controlled by attackers. Therefore, uninitialized-use vulnerabilities can be readily exploited.

To prevent the exploitation of uninitialized pointer uses, we propose POINTSAN, a compiler-based mitigation that initializes potentially unsafe pointer-type fields. Our mitigation is inspired by the observation that uninitialized-use exploits typically control an uninitialized pointer to achieve arbitrary read/write/execution. By zero-initializing pointer-type fields in an allocated object, we can prevent an adversary from controlling these pointers. Since the memory page at the address zero is not accessable in modern OS kernels, zero-initialization becomes a safe prevention operation. More specifically, we perform an intra-procedural analysis that identifies potentially uninitialized pointer-type fields and instrumentation that zero-initializes the identified pointer-type fields. POINTSAN imposes almost no performance overhead to the secured kernels.

1.2.2 Retrofitting System Designs Against Information Leaks

System designers often prioritize performance and flexibility over security; hence, many system designs are insecure, which is another root cause of information leaks. Our second

way to prevent information leaks is redesigning insecure system mechanisms. A well-known insecure system design causing information leaks is the ASLR-violating worker process forking scheme in daemon servers (e.g., web servers). To prevent information leaks caused by such an insecure design, we propose the runtime re-randomization mechanism.

1.2.2.1 RUNTIMEASLR*: Re-Randomizing Memory Layout at Runtime*

Existing techniques for memory randomization such as the widely deployed ASLR perform a single, per-process randomization that is applied before or at a process' load-time. The efficacy of such upfront randomizations crucially relies on the assumption that an attacker has only one chance to guess the randomized address, and that this attack succeeds only with a very low probability. Recent research results have shown that this assumption is not valid in many scenarios, e.g., daemon servers fork child processes that inherent the state—and if applicable: the randomization—of their parents, and thereby create clones with the same memory layout. This enables the so-called clone-probing attacks where an adversary repeatedly probes different clones in order to increase its knowledge about their shared memory layout.

We propose RUNTIMEASLR—the first approach that prevents clone-probing attacks without altering the intended semantics of child process forking. The work makes the following three contributions. First, we propose a semantics-preserving and runtime-based approach for preventing clone-probing attacks by re-randomizing the address space of every child after fork() at runtime while keeping the parent's state. We achieve this by devising a novel, automated pointer tracking policy generation process that has to be run just once, followed by a pointer tracking mechanism that is only applied to the parent process. Second, we propose a systematic and holistic pointer tracking mechanism that correctly identifies pointers inside memory space. This mechanism constitutes the central technical building block of our approach. Third, we provide an open-source implementation of our approach based on Intel's Pin on an x86-64 Linux platform, which supports COTS server binaries

directly. Although it takes a longer time for RUNTIMEASLR to start server programs, RUNTIMEASLR imposes almost no runtime performance overhead to the worker processes that provide actual web services.

1.2.3 Protecting Sensitive Data from Information Leaks

It is unfortunate that we could not fix all information-leak vulnerabilities or insecure system designs because of the complexity of systems and the continuously introduced system features. For example, we cannot even identify all side channels that may leak information, thus cannot prevent all information leaks. As a response to this problem, we instead strive to protect certain sensitive data in the presence of information leaks. Towards this goal, we propose preventing code-pointer leaks to defeat code-reuse attacks [164], and a replicated execution-based framework to detect memory disclosures.

1.2.3.1 ASLR-GUARD*: Stopping Code-Pointer Leaks*

A general prerequisite for a code-reuse attack is that the attacker needs to locate code gadgets that perform the desired operations and then direct the control flow of a vulnerable application to those gadgets. ASLR attempts to stop code-reuse attacks by making the first part of the prerequisite unsatisfiable. However, research in recent years has shown that this protection is often defeated by commonly existing information leaks, which provides attackers clues about the whereabouts of certain code gadgets.

We propose ASLR-GUARD, a novel mechanism that completely prevents the leaks of code pointers. The main idea behind ASLR-GUARD is to provide a secure storage for code pointers and encode the code pointers when they are treated as data. ASLR-GUARD can either prevent code pointer leaks or render their leaks harmless. That is, ASLR-GUARD makes it impossible to overwrite code pointers with values that point to or will hijack the control flow to a desired address when the code pointers are dereferenced. ASLR-GUARD completely stops code-pointer leaks and can resist against recent sophisticated code-reuse attacks. Further, ASLR-GUARD imposes almost no runtime overhead ($< 1\%$) to the secured

programs. Therefore, ASLR-GUARD is very practical and can be applied to secure many applications.

1.2.3.2 BUDDY*: Detecting Memory Disclosures with Replicated Execution*

Memory disclosure is an attack primitive that has been used to bypass widely deployed security mechanisms such as ASLR (e.g., the JIT-ROP attack [167]) and to leak sensitive data such as private keys (e.g., the HeartBleed attack [62]). The most common way to disclose memory is exploiting memory errors such as out-of-bound read, use-after-free, and uninitialized use. Since preventing these memory errors is hard, and the number of reported memory errors is still increasing, memory disclosure has become a common but hard-to-stop security threat.

We propose a new framework, BUDDY, to comprehensively detect memory disclosures with replicated execution. The intuition behind BUDDY is that, by seamlessly maintaining two running instances of a program and diversifying only the sensitive data, we can detect any leaks of such data, as doing so will result in the two instances outputting different values. BUDDY detects memory disclosures without the need of performing the error-prone and expensive data-flow tracking. Further, BUDDY's approach can support COTS binaries. We propose multiple diversification schemes for both data and memory layout, and to design an efficient replicated execution engine. By applying BUDDY to poplar programs, including Apache web server, Nginx web server, PHP server, OpenSSL, Lighttpd, and the SPEC benchmarks, we extensively evaluated its reliability, security, and performance.

1.3 Thesis Contributions

In summary, this thesis makes the following contributions to securing software systems.

- **General defense against system attacks.** We identify that, in modern systems, preventing information leaks can be a general and efficient approach to defeating both data leaks and control attacks.

- **Study of information leaks.** We study information leaks in computer systems and provide insights into their causes and how to prevent them.
- **Discovery of new threats.** We identify a fundamental security problem in compilers, which causes common information leaks. We also revisit the security impact of using uninitialized pointers.
- **Three ways to prevent information leaks.** We propose general ways to eliminate root causes of information leaks and to protect sensitive data, even in the presence of information leaks.
- **Novel defense mechanisms and open source.** We design multiple automated and practical defense mechanisms to eliminate, detect, and prevent information leaks. Most prototype implementation has been open-sourced.

CHAPTER II

ELIMINATING INFORMATION-LEAK VULNERABILITIES

The most common cause of information leaks is that software systems have a significant number of vulnerabilities such as buffer over-read, use-after-free, and uninitialized read. According to the U.S. National Vulnerability Database [57], the number of information-leak is increasing. In particular, information-leak vulnerabilities have become more prevalent than other types of vulnerabilities such as buffer overflow in the Linux kernel. Attackers have exploited information-leak vulnerabilities to leak sensitive data or leak randomized value to break defense mechanisms such as ASLR. Hence, an intuitive and effective way to prevent information leaks is to eliminate information-leak vulnerabilities. Towards this goal, we have studied information-leak vulnerabilities in the Linux kernel and found that more than half of them are caused by uninitialized uses including uninitialized data read and uninitialized pointer dereferencing. While buffer over-read and use-after-free vulnerabilities are well studied, practical uninitialized use prevention is largely an uncharted territory. In this section, we propose two techniques, UNISAN and POINTSAN, to eliminate information-leak vulnerabilities caused by uninitialized uses in OS kernels. UNISAN completely eliminates the ones caused by uninitialized data reads; POINTSAN first revisits the security impact of uninitialized pointer dereferencing and then eliminates it in a practical manner.

2.1 UNISAN*: Eliminating Uninitialized Data Leaks*

As the de facto trusted computing base (TCB) of computer systems, the operating system (OS) kernel has always been a prime target for attackers. By compromising the kernel, attackers can escalate their privilege to steal sensitive data in the system and control the whole computer. There are three main approaches to launching privilege escalation attacks: (1) direct code injection attacks; (2) `ret2usr` attacks [91]; and (3) code reuse attacks [164].

DEP (Data Execution Prevention) protection has been deployed to defeat traditional code injection attacks. Intel and ARM have recently introduced new hardware features (e.g., SMEP and PXN) to prevent `ret2usr` attacks [89]. As such, code reuse attacks are becoming more prevalent. A general direction to defeat code reuse attacks is randomization, e.g., kernel address space layout randomization (kASLR) [42] and StackGuard [44] have been adopted by the latest kernels. kASLR aims to prevent attackers from knowing where the code gadgets are, and StackGuard [44] aims to prevent attackers from corrupting return addresses.

By design, the effectiveness of kASLR and StackGuard completely relies on the confidentiality of the randomness—leaking any randomized pointer or the stack canary will render these mechanisms useless. Unfortunately, information-leak vulnerabilities are common in OS kernels. For example, Krause recently found 21 information leaks in the Linux kernel [95]. According to CVE Details [57], kernel information-leak vulnerabilities have not only become more prevalent than buffer overflow vulnerabilities, but the number of kernel information leaks is also increasing with continuously introduced new features. Moreover, the leaked kernel memory may also contain other sensitive data, such as cryptographic keys and file caches. For these reasons, preventing kernel information leak is a pressing security problem.

There are four main causes of information leaks: uninitialized data read, buffer over-read, use-after-free and logic errors (e.g., missing privilege check). Among these root causes, uninitialized read is the most common one. According to a Linux kernel vulnerabilities survey [33], 37 information-leak vulnerabilities were reported from January 2010 to March 2011, 28 of which were caused by uninitialized data read. Similarly, we analyzed the causes of the kernel information leaks reported between January 2013 and May 2015. Our study also revealed that about 60% kernel information leaks are caused by uninitialized data read (Figure 1). However, while many memory-safety techniques [129, 131, 132, 160] have been proposed to prevent buffer over-read and use-after-free, practical prevention of uninitialized

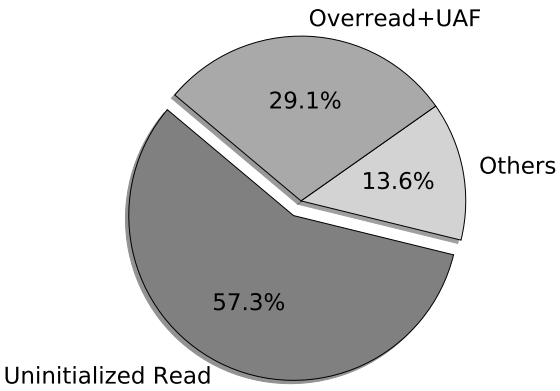

Figure 1: The root causes of kernel information leaks reported between January 2013 and May 2016: uninitialized data read, spacial buffer over-read + use-after-free, and others. Most leaks are caused by uninitialized data read.

data leaks is still an open problem.

Preventing uninitialized data leaks is challenging for three reasons. First, since data write and read are frequent in programs, detecting uninitialized data reads by tracking these operations will always introduce unacceptable performance overhead. For example, MemorySanitizer [172] and kmemcheck [138] check every memory read and write to detect uninitialized data reads, thus incurring a performance overhead of more than three times. Second, uninitialized data reads are actually quite common in programs. Specifically, compilers often introduce padding bytes in data structures to improve performance. These padding bytes are usually uninitialized, but as long as the uninitialized data is not used (e.g., dereferenced as pointer or leaked), its access does not cause any problem. Since the padding is introduced by the compiler, developers are usually not aware of the potential data leaks—they need to be convinced such problems exist before they will fix the programs (see §2.1.1 for more details). Third, uninitialized data leaks often occur across multiple procedure boundaries—the uninitialized data is always passed to leaking functions (e.g., `copy_to_user`); hence intra-procedure analysis based detections (e.g., `-Wuninitialized` provided by compilers) cannot catch uninitialized data leaks.

In addition to tracking every data read and write, researchers have attempted an alternate approach—*force initializing*. For example, PaX's `STACKLEAK` plugin [175] clears the used

kernel stack when the control is transferred back to user space, which effectively prevents data leaks between syscalls. However, `STACKLEAK` cannot prevent the leaking of uninitialized data generated during the same syscall. Also, this can introduce a significant performance overhead (see Table 5). Split kernel [96] instead clears stack frame whenever it is allocated (i.e., a function is called). Split kernel provides stronger security, but its performance overhead is even more significant. Peiró et al. [148] proposed using model checking to detect kernel stack allocations that have never been `memset` or assigned. However, this approach has obvious limitations. For example, since it neither tracks the propagation of uninitialized data nor handles partial initialization, it has high false negative and false positive rates. Moreover, none of these approaches can handle kernel heap leaks.

In this work, we propose a novel mechanism, UNISAN (Uninitialized Data Leak Sanitizer), to prevent kernel leaks caused by uninitialized data reads. Similar to `STACKLEAK`, UNISAN is an automated compiler-based approach; that is, it does not require manual modifications to the source code and can transparently eliminate leaks caused by data structure padding or improper initialization. At the same time, UNISAN also overcomes all the aforementioned limitations of previous force-initialization approaches. More specifically, UNISAN leverages an inter-procedural static analysis to check *(1) whether an allocated object ever leaves kernel space, and (2) whether the allocated object is fully initialized along all possible execution paths to the leak point*. The analysis is conservative—as long as there is one byte of an allocated object that cannot be proved to have been initialized in any possible execution path before leaving the kernel space, the corresponding allocation is considered *unsafe*; hence UNISAN has no false negatives. It also handles both stack and heap allocations. At the same time, to improve the precision and thus minimize false positives, UNISAN's analysis is fine-grained, which tracks each byte of an allocated object in a flow-sensitive and context-sensitive manner. Once UNISAN detects an unsafe allocation, it then instruments the kernel to zero the uninitialized portion of the objects allocated by them. In this way, UNISAN can completely prevent kernel information leaks caused by

uninitialized reads. Note that by being conservative, UNISAN may still have false positives; however, zero-initializing allocated objects that will never be leaked will not break the semantics of the kernel, but will just introduce unnecessary performance overhead. Because UNISAN's instrumentation is semantic-preserving, robustness is guaranteed.

We have implemented UNISAN based on the LLVM compiler [112]. UNISAN consists of two components. The first is the *unsafe allocation detector*, which conservatively reports all potentially unsafe allocations. The second is the *unsafe allocation initializer*, which zeros the uninitialized memory by inserting zero-initialization, `memset`, or changing the allocation flags.

We have applied UNISAN to the latest mainline Linux and Android kernels to evaluate the effectiveness and efficiency of UNISAN in preventing kernel leaks. For effectiveness evaluation, we first tested UNISAN over 43 recently discovered kernel information-leak vulnerabilities resulting from uninitialized reads. UNISAN successfully detected and prevented all these known vulnerabilities. Moreover, the unsafe allocation detector of UNISAN has identified many new uninitialized data leak vulnerabilities in the latest Linux kernel and Android kernel, 19 of which have been confirmed by the Linux maintainers and Google.

We also measured UNISAN's performance impacts on kernel operations, server programs, and user-space programs using multiple benchmarks, including LMbench [125], ApacheBench, Android benchmarks, and the SPEC CPU Benchmarks. The evaluation results show that UNISAN incurs a negligible performance overhead (less than 1% in most cases) and is thus much more efficient than existing solutions (e.g., `STACKLEAK`).

We believe that since UNISAN is robust, effective, and efficient, it is ready to be adopted in practice to prevent uninitialized data leaks. In summary, we make the following contributions:

- **Survey of kernel information leaks**: We studied all the kernel information-leak vulnerabilities reported between January 2013 and May 2016. We analyzed their root causes and corresponding defenses. We also discussed with kernel developers about

how to prevent information leaks caused by data structure padding.

- **Development of new protection mechanism**: We designed and implemented UNISAN, an automated, compiler-based scheme, to eliminate kernel information leaks caused by uninitialized data reads, which is the main cause of kernel leaks. UNISAN has been successfully applied to the latest mainline Linux and Android kernels (yet not limited to kernels). UNISAN is a practical, ready-to-use security protection scheme, and has been open-sourced for broader adoption.

- **Discoveries of new vulnerabilities**: During our evaluation, UNISAN discovered many previously unknown information-leak vulnerabilities in the latest Linux and Android kernels, 19 of which have been confirmed by Linux maintainers and Google.

2.1.1 An Analysis of Kernel Information Leaks

In this section, we provide a background on kernel information leaks, including their security implications and root causes. We then discuss uninitialized data reads and how such vulnerabilities should be fixed from developers' perspectives.

2.1.1.1 Kernel Information Leaks

Kernel information-leak vulnerabilities can cause severe security consequences. First, with the deployment of kASLR [42] and stack canary [44], a general prerequisite for many attacks (e.g., code reuse attack) is learning the randomized addresses and canary, which can be accomplished by exploiting kernel information-leak vulnerabilities. Further, as the TCB of the whole system, the OS kernel also has access to many other types of sensitive information, such as encryption keys, file cache, and remaining data of terminated processes, etc. For performance reasons, memory pages allocated to store such information may not be cleared when they are released to the kernel. As a result, kernel information-leak vulnerabilities also allow attackers to access such sensitive information. For example, [185] showed that an uninitialized data leak is used to leak the entropy source for `srandom`. In short, it is critical

to prevent kernel information leaks.

Kernel information leaks are also very common and have various causes. According to a previous study [33] of Linux kernel vulnerabilities discovered between January 2010 and March 2011, kernel information leak was ranked the second most common vulnerability, which is even more common than buffer overflow vulnerability. Specifically, of the total of 37 leak vulnerabilities, 28 were caused by uninitialized reads, 7 were caused by buffer over-read, and two by other miscellaneous causes.

Since the aforementioned study [33] is already five-year old, we conducted another study of kernel information-leak vulnerabilities reported between January 2013 and May 2016 [57], which contained 103 leaks in total. The result is shown in Figure 1. The majority of kernel leaks are caused by uninitialized reads. Buffer over-read is also a common cause, in which the size of reading is not properly checked. Use-after-free bugs may be exploited to leak the data of newly allocated objects or manipulate the size and address of the read. Other causes include missing permission check, race condition, or other logic errors. Since uninitialized read is the most common cause of kernel leaks, our work focuses on preventing uninitialized data leaks.

2.1.1.2 Uninitialized Data Leaks

An uninitialized data leak occurs when an allocated stack or heap object is not properly initialized when being copied to the outside world (e.g., to user space, network, or file systems). If the memory occupied by the object is used to store sensitive data (e.g., addresses), attackers can exploit this to leak such information. Note that *using* uninitialized memory is a type of memory-safety error and can lead to undefined behavior. For this reason, compiler features such as the `-Wuninitialized` option in GCC generate warnings when variables are used without proper initialization. However, such compiler features employ only an *intra-procedure* analysis and cannot handle many common cases (e.g., reading the uninitialized data through its pointer). Most uninitialized data leaks happen across multiple

function boundaries (e.g., calling `copy_to_user` in Linux), so they can be identified only using *inter-procedural* analysis. Also, data can propagate through various channels (e.g., network or file systems). In short, existing compiler checks are not effective in finding uninitialized data leaks in large-scale, sophisticated programs such as the Linux kernel. Moreover, uninitialized data reads may not be harmful if they are not dereferenced or leaked; reporting all of them will burden or even annoy developers. The following examples demonstrate the common causes of uninitialized data leaks and why existing compiler features cannot detect them.

Missing Element Initialization. The simplest and most common case of an uninitialized data leak is when the developers fail to properly initialize all fields of an object or memory of a buffer. Figure 2 shows a real kernel leak vulnerability in the x25 module. Specifically, the object `dte_facilities` is supposed to be properly initialized in `x25_negotiate_facilities`. However, six fields are still not initialized. The object is then propagated to the external heap object `makex25` and finally leaked to userland in another function `x25_ioctl`. As we can see, detecting such leaks would require sophisticated inter-procedural data-flow analysis, which is not available during normal compilation.

Figure 3 shows another real kernel leak vulnerability caused by program design and developers. The 6-bytes buffer `mac_addr` is allocated on stack. It is supposed to be initialized in the `dump_station` function before it is sent out. However, since `dump_station` is an interface function that has different implementations for different protocols written by different programmers, it is hard to ensure the object is consistently initialized. Specifically, this buffer is not initialized in the implementation of the `wilc1000` module. As this uninitialized data is actually never *used* by the kernel, it is simply copied outside the kernel boundary; from the perspective of compiler, these cases are not errors. Moreover, `dump_station` is a function pointer (i.e., dereferenced by an indirect call), tracking which requires a complete call graph that is not provided by current compiler features.

Data Structure Padding. Besides developer mistakes, a more interesting source of

```
/* File: net/x25/af_x25.c */
int x25_rx_call_request(struct sk_buff *skb,
                        struct x25_neigh *nb,
                        unsigned int lci) {
★  struct x25_dte_facilities dte_facilities;
   /* some fields of dte_facilities are not initialized */
!  x25_negotiate_facilities(..., &dte_facilities);
   ...
   /* passed to the external */
   makex25->dte_facilities= dte_facilities;
   ...
}
static int x25_ioctl(struct socket *sock,
                     unsigned int cmd,
                     unsigned long arg) {
   ...
   /* leak uninitialized fields of dte_facilities */
⊙  copy_to_user(argp, &x25->dte_facilities,
                sizeof(x25->dte_facilities));
   ...
}
```

Figure 2: New kernel leak in the `x25` subsystem—six fields of `dte_facilities` are not initialized and leaked in another function `x25_ioctl`. ⋆ denotes memory allocation, ! marks incorrect initialization, ⊙ notes a leaking point.

```
/* File: net/wireless/nl80211.c */
static int nl80211_dump_station(struct sk_buff *skb,
                                struct netlink_callback *cb) {
⋆  u8 mac_addr[ETH_ALEN]; /* ETH_ALEN = 6 */
   ...
!  err = rdev->ops->dump_station(\
          &rdev->wiphy,
          wdev->netdev, sta_idx, mac_addr, &sinfo);
   /* mac_addr is uninitialized but sent out via nla_put()
    * inside nl80211_send_station() */
⊙  if (nl80211_send_station(skb, NL80211_CMD_NEW_STATION,
                           NETLINK_CB(cb->skb).portid,
                           cb->nlh->nlmsg_seq, NLM_F_MULTI,
                           rdev, wdev->netdev, mac_addr,
                           &sinfo) < 0)
     goto out;
   ...
}
```

Figure 3: New kernel leak in the `wireless` subsystem—the whole 6-bytes array `mac_addr` is not initialized but sent out. ⋆ denotes memory allocation, ! marks incorrect initialization, and ⊙ notes a leaking point.

uninitialized data leaks is compiler-added data structure padding. In particular, when modern processors access a memory address, they usually do this at the machine word granularity (e.g. 4-bytes on a 32-bits systems) or larger. So if the target address is not aligned, the processor may have to access the memory multiple times to perform shifting and calculating to complete an operation, which can significantly degrade the performance [151]. Moreover,

```
1  /* File: drivers/usb/core/devio.c */
2  struct usbdevfs_connectinfo {
3    unsigned int devnum;
4    unsigned char slow;
5    /* 3-bytes padding inserted for alignment */
6  };
7  static int proc_connectinfo(struct usb_dev_state *ps,
8                              void __user *arg) {
9  ⋆  struct usbdevfs_connectinfo ci = {
10     .devnum = ps->dev->devnum,
11     .slow = ps->dev->speed == USB_SPEED_LOW
12   };
13
14   /* sizeof(ci) == 8, but only 5 bytes are initialized */
15 ⊙  if (copy_to_user(arg, &ci, sizeof(ci)))
16     return -EFAULT;
17   return 0;
18 }
```

Figure 4: New kernel leak caused by compiler padding. Object `ci` contains 3-bytes padding at the end, which is copied to userland. Note that ⋆ denotes memory allocation and partial initiation, and ⊙ notes a leakage point.

different instructions or architectures could also have their own alignment requirements. For instance, SSE instructions on the x86 architecture require the operands to be 16-bytes aligned, and ARM processors always require memory operands to be 2-bytes or 4-bytes aligned. For these reasons, compilers usually add padding within a data structure so that fields are properly aligned.

The problem is that as these padding bytes are not visible at the programming language level, and they are not initialized even when all fields have been properly initialized in the code written by developers and hence may lead to information leaks. Figure 4 illustrates a kernel leak in the USB module caused by data structure padding. In this case, although developers have explicitly initialized all fields of the stack object `ci`, because of its last 3-byte padding, information can still be leaked when `copy_to_user` is invoked to copy the whole object to user space. To detect such leaks, byte-level analysis is required; object-level or even field-level (i.e., field-sensitive) analyses are still not fine-grained enough to catch such leaks.

To summarize, leaking padding bytes is a serious problem for three reasons. First, it is prevalent. Compilers frequently introduce padding for better performance. Padding is

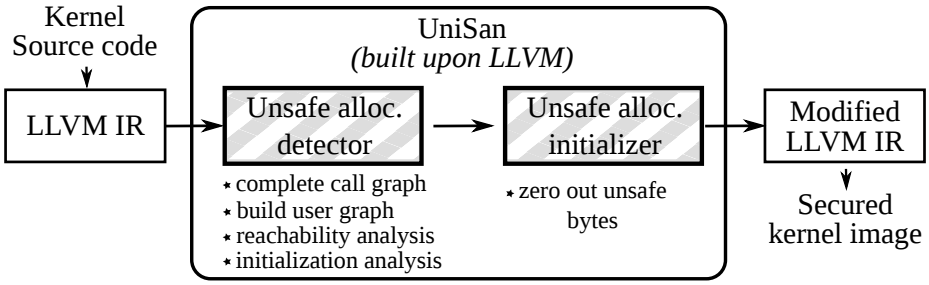

Figure 5: Overview of UNISAN's workflow. UNISAN takes as input the LLVM IR (i.e., bitcode files) and automatically generates secured IR as output. This is done in two phases: unsafe allocation detection and zero initialization.

even more prevalent when porting programs from 32-bits to 64-bits platforms. Second, inter-procedural and byte-level analysis is required to detect such leaks. Third, it is often not visible to developers. Padding bytes can be leaked even developers have correctly initialized all fields.

After we submitted patches for these vulnerabilities, the compiler-injected padding issue was discussed extensively by the Linux community and became a major concern to the Linux kernel developers because, from the developers' perspective, they have properly initialized the data structures and this type of leak is hardly visible even to skilled programmers. On the other hand, from the compilers' perspective, they have the benefit of *not* proactively initializing such padding regions to achieve better performance, because this design decision can be independently made by each compiler according to the C/C++ specification. However, considering its severity, prevalence and more importantly, its non-trivial nature to developers, *we, as well as many kernel maintainers, urge the incorporation of* UNISAN*'s approach to solve this problem at compiler level, perhaps as an extra option to compilers.*

2.1.2 Overview of UNISAN

In this work, we focus on *preventing* kernel information leaks caused by uninitialized data reads. UNISAN achieves this goal via a two-phase process (Figure 5). In the first phase, UNISAN analyzes the kernel and conservatively identifies all potential unsafe allocations whose objects may leave the kernel space without having been fully initialized. Then in

the second phase, UNISAN instruments the kernel to automatically initialize (zero out) the detected unsafe allocations.

2.1.2.1 Problem Scope

The protection target of UNISAN is OS kernels. Since UNISAN performs analysis over source code, we assume that the source code of the kernel is available. We assume the attackers do not have the kernel privilege (e.g., through loading a malicious kernel driver), but they can be either local or remote. The goal of the attackers is to leak sensitive data (e.g., addresses, cryptographic keys, file content) in the kernel space to the external world (e.g., userland and network). After gathering the information, the attacker can launch more attacks, such as privilege escalation and phishing.

As discussed in §2.1.1, kernel information leaks have multiple root causes: uninitialized data read, buffer over-read, use-after-free, data race, logical errors, etc. In this work, we focus on uninitialized data read because a majority of kernel leaks are caused by this particular type of vulnerabilities and there is no practical solution yet; whereas other types of vulnerabilities can be addressed by existing techniques and are thus out-of-scope. For instance, buffer over-read and use-after-free bugs can be prevented by memory safety techniques [129, 131, 132, 160]. Data race can be detected by [64]. Logical errors (e.g., missing permission check) are relatively rare and can be identified by semantic checking techniques [13, 126].

2.1.2.2 The UNISAN *Approach*

There are multiple candidate solutions for preventing uninitialized data leaks. The first is zeroing the memory when it is deallocated. Since we cannot know when the deallocated memory will be allocated and used in the future, we have to conservatively zero all deallocated stack and heap objects, which can introduce a significant performance overhead. More importantly, deallocation is not always paired with allocation, such as the case of memory leak, thus introducing false negatives. The second is dynamically tracking the

status of every byte in the object, so that we can know exactly if any uninitialized bytes are leaving the kernel space. MemorySanitizer [172] and kmemcheck [138] are based on this approach. However, while they are effective and able to detect general uninitialized data uses, their 3x performance overhead is too high to be used for runtime prevention. The third is selectively zeroing the allocated memory that is detected as unsafe by static analysis. After assessing the effectiveness and performance of these approaches, we chose the third approach—initializing only the unsafe allocations.

Figure 5 illustrates the approach of UNISAN. Specifically, UNISAN takes as input the LLVM IR (i.e., bitcode files) compiled from the kernel source code, upon which the analysis and instrumentation are performed. Given a stack or heap allocation, UNISAN leverages static data-flow (taint) analysis to check whether this allocation can reach the pre-defined sinks, such as `copy_to_usr` and `sock_sendmsg`. Along the propagation path, UNISAN also tracks the initialization status of each byte of the allocation. If any byte of the allocation reaches the sink without having been initialized in any possible execution path, UNISAN considers it unsafe. After collecting all unsafe allocations, UNISAN instruments the IR with initialization code right after them. For stack allocation, UNISAN inserts a `memset` or zero-initiailzation to initialize it. For heap allocation, UNISAN adds the `__GFP_ZERO` flag to the allocation functions (e.g., kmalloc). Finally, by assembling and linking the instrumented bitcode files, UNISAN generates the secured kernel image.

2.1.3 Design

In this section, we present the design of UNISAN. We first describe the design of *unsafe allocation detector*, including how we generate the complete call graph, perform reachability analysis, and track the initialization status of allocations. Then we describe how *unsafe allocation initializer* instruments the kernel to generate secured kernel images.

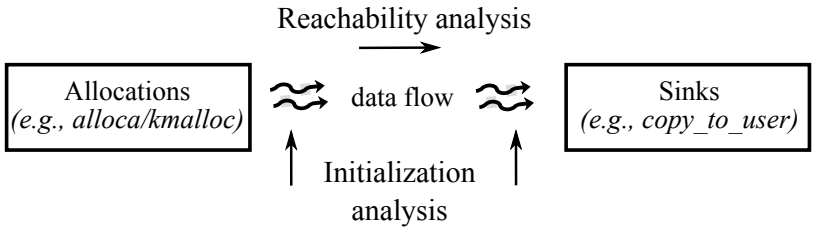

Figure 6: Unsafe allocation detector incorporates a reachability analysis and an initialization analysis to check whether any byte of the allocation is uninitialized when leaves the kernel space.

2.1.3.1 Detecting Unsafe Allocations

Our unsafe allocation detection is essentially a static taint analysis that incorporates two orthogonal analyses: object initialization analysis and sink reachability analysis, as shown in Figure 6. The initialization analysis checks which bytes of the object will be initialized along the paths from allocation to sinks, i.e., which bytes will be assigned with other values. The sink reachability analysis checks which bytes of the object will leave the kernel space along the paths from allocation to sinks, i.e., being passed to sinks. The initialization analysis and reachability analysis are then integrated to detect unsafe bytes—a byte is *unsafe* if it is uninitialized when it leaves kernel space.

The workflow of the detection is as follows. Given the bitcode files of the target program, it first builds a complete and global call graph. Then it parallelly performs the reachability analysis and initialization analysis for each allocation to detect the unsafe bytes. Our analysis is flow-sensitive in that the order of the uses of the allocation is maintained with a dedicated *user-graph* (that will be elaborated in Figure 2.1.3.1); context-sensitive in that function calls are followed with callsite-specific context; and field-sensitive in that it performs fine-grained tracking for each byte of the allocation. However, to avoid the path explosion problem and make our analysis scalable to the whole kernel, our analysis is path-insensitive.

Defining Sources and Sinks. Both the reachability analysis and initialization analysis track the "taint" status of the allocated bytes from the allocation site to the sink functions. As the first step, we need to pre-define the sources (i.e., allocations) and sinks (i.e., data-leaking functions).

For stack, all objects are allocated by `AllocaInst` (i.e., an instruction to allocate memory on the stack). By handling this instruction, we are able to find all tracking sources on stack. Heap objects can be allocated in many ways. In our current implementation, we include only the standard allocator from SLAB, namely, `kmalloc` and `kmem_cache_alloc`. These heap allocators accept a flag parameter. If the flag contains `__GFP_ZERO` bit, the allocated memory will be initialized with zero. In UNISAN, we track only heap allocations without the `__GFP_ZERO` flag. Please note that although UNISAN currently does not include custom allocators, it can be easily extended to support them once developers denote the function name and allocation flags (i.e., create the source signature).

Under the threat model of UNISAN, any function that may send kernel data to userland, network, or file is classified as a sink function. In UNISAN, we use two policies to generally define the sinks. We first empirically define a list of known sink functions, based on our study of previous kernel leaks. For example, `copy_to_user` copies data to userland; `sock_sendmsg` sends data to network, and `vfs_write` writes data to files. Although there are various implementations for file writing (for different file systems) and message sending (for different protocols), `vfs_write` and `sock_sendmsg` are the uniformed interfaces, so we can generally catch the sink functions by annotating these functions.

Clearly, there are more sink functions that are not covered by the first step. To eliminate false negatives introduced by an incomplete sink list, we utilized three conservative rules to generally cover additional sinks. These rules are defined based on the fact that under our recursive tracking algorithm (Figure 7), *for any data to leave kernel space, it will always be stored to a non-kernel-stack location (or non-`AllocaInst` to be specific)*, so once we cannot determine that the destination of an store operation is on kernel stack, we treat it as a sink.

- **Rule 1**: A `StoreInst` (i.e., an instruction for storing to memory) is a sink if the destination is not allocated by an `AllocaInst` in kernel;
- **Rule 2**: A `CallInst` (i.e., an instruction for calling a function) is a sink if the called value is inline assembly that is not in the whitelist (Figure 2.1.3.1);

```
/* Unsafe allocation detection algorithm */
UnsafeAllocDetection(Module) {
    for (Alloc in Module) {
        /* user relationships are maintained */
        MergedUnsafeBytes = Array();
        UserGraph = BuildUserGraph(Alloc);
        NextUsers = GetFirstUser(UserGraph);
        RecursiveTrackUsers(Alloc,NextUsers,MergedUnsafeBytes);
        if (IsNotEmpty(MergedUnsafeBytes))
            Report(Alloc, MergedUnsafeBytes);
    }
}
RecursiveTrackUsers(Alloc, NextUsers, MergedUnsafeBytes) {
    /* terminate if all bytes have been inited or sinked */
    if (AllInitied(Alloc) || AllSunk(Alloc))
        return;
    UnsafeBytes = Array();
    for (User in NextUsers) {
        /* Next users of User are recursively tracked */
        if (IsLoadInst(User))
            RecursiveTrackLoad(User, UnsafeBytes);
        else if (IsStoreInst(User))
            RecursiveTrackStore(User, UnsafeBytes);
        else if (IsCallInst(User))
            RecursiveTrackCall(User, UnsafeBytes);
        else if (IsGetElementPtr(User))
            RecursiveTrackGEP(User, UnsafeBytes);
        ...
        /* Unrecognized cases */
        else
            /* assume remaining uninitialized bytes unsafe */
            UnsafeBytes += GetUninitializedBytes(Alloc);
        /* Conservatively merge (union) all unsafe bytes */
        MergedUnsafeBytes += UnsafeBytes;
    }
}
```

Figure 7: The pseudo-code of the recursive tracking algorithm for the unsafe allocation detection.

- **Rule 3**: A `CallInst` is a sink if the called function's body is empty (i.e., not compiled into LLVM IR).

Building Global Call-Graph. Since UNISAN's analysis is inter-procedural, global call graph is required. To eliminate false negatives, UNISAN must conservatively identify all potential targets of indirect calls. To this end, we first collect the address-taken functions, and use the type-analysis-based approach [135, 177] to find the targets of indirect calls. That is, as long as the type of the arguments of an address-taken function matches with the callsite of the indirect call, we assume it is a valid target. Note that we also assume universal pointers (e.g., `char *`, `void *`) and an 8-bytes integer can match with any type.

Recursive Detection Algorithm. With the global call-graph, we conduct the unsafe

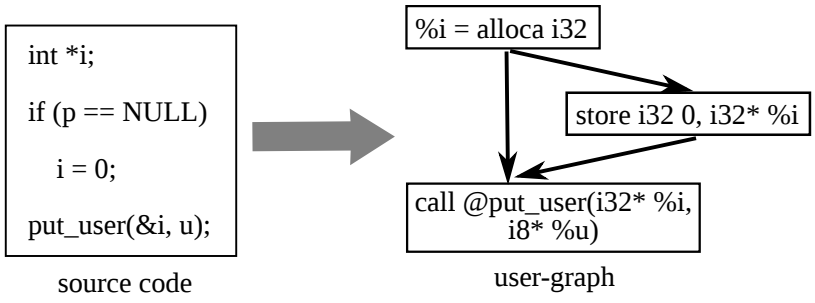

Figure 8: A simple user-graph example.

allocation detection. The algorithm of the detection is shown in Figure 7. In short, given an allocation in a module, we first build its user-graph (that will be elaborated in Figure 2.1.3.1). After that, we recursively keep track of which bytes of the allocated object have been initialized and which bytes have reached sink functions, by traversing the user-graph. Different users are handled properly. If there are any corner cases that are not recognized before, we conservatively assume the bytes being tracked are unsafe. All unsafe bytes in different paths are concatenated together. That is, a byte is assumed unsafe as long as it is unsafe in any possible path.

Building User-Graph. In LLVM IR, a *user* of a value is an instruction or expression that takes the value as an operand. The unsafe allocation detection is designed to be flow-sensitive, which requires considering the order of users. Given the being tracked value (e.g., the allocated value), LLVM framework only tells us all the users but not the relationships (e.g., sequential and parallel relationships) among them. To maintain the relationships of the users, we build the user-graph for the tracked value. Figure 8 shows a simple user-graph example. Instructions that do not use the tracked value will not show up in the graph. More specifically, we first put all the users in the corresponding basic blocks. Users in the same basic block are always in a sequential order. Then we use the `DominatorTree` pass of LLVM to understand the relationships among the involved basic blocks. With this information, we chain all the users together into a user graph. When an alias of the tracked value is generated (e.g., by the `CastInst` and `GetElementPtr` instructions), all the users of the alias will be seamlessly merged into the user-graph as well.

Fine-Grained Status Tracking. UNISAN performs unsafe allocation detection in a fine-grained manner (i.e., byte-level analysis). There are several advantages of performing the analysis at byte granularity. First, due to the compiler's padding, initializing all fields of an object does not guarantee that all bytes are initialized (Figure 4). Therefore, byte-level analysis is necessary to detect the uninitialized padding bytes. Second, `union` is widely used in kernel data structures and byte-level tracking can help resolve the issue introduced by field alias. Finally, byte-level detection can precisely filter out safe bytes, so that the instrumentation module can selectively initialize only the unsafe ones, thus further reducing the runtime performance overhead.

To perform the byte-level analysis, we use a buffer to record the initialization and sinking status of every byte in a tracked allocation. Whenever an object is allocated, a buffer of the same size is created. Currently, we only use two bits of the corresponding byte in the buffer to represent the initialization and sinking status of each byte in the original object. To keep track of which bytes are being accessed, at every `GEPOperator` (an instruction or expression for type-safe pointer arithmetic to access elements of arrays and structs) node in LLVM IR, we calculate the offset (into the base of the object) and the size of the obtained element. In our current design, we do not perform range analysis, so if any of the indices of the `GEPOperator` node are not constants, we cannot statically calculate the resulting element. In this case, we conservatively treat all bytes of the allocation as uninitialized and pause the initialization analysis—as long as the reachability analysis determines that the allocation can leave the kernel space, we assume it is unsafe. Since non-constant indices are not common in `GEPOperator` node, our byte-level analysis works well in most cases.

Eliminating False Negatives. As a prevention tool, UNISAN aims to eliminate potential false negatives. Clearly, aggressively initializing all stack and heap allocations can guarantee no false negative; however, such a naive approach will introduce unnecessary performance overhead, since most allocations will either never leave kernel space or be properly initialized. To make UNISAN more efficient, our principle is to eliminate as many false positives as

possible while ensuring no false negative; and whenever we encounter an undecidable problem or an unhandled corner case, we always sacrifice the detection accuracy and assume the tracked allocation is unsafe. In this subsection, we summarize cases that may introduce false negatives and describe how we handle them.

Complete call graph. LLVM's built-in call graph pass does not find callees of indirect calls. As described in Figure 2.1.3.1, we adopt a conservative type-analysis to find indirect call targets to build a complete and global call graph.

Conservative path merging. There are often many paths from the allocation site to the sink points. And in different paths, the allocated object can be initialized differently. In LLVM IR, the most common cases that can introduce multiple paths include: (1) load and store instructions. These instructions copy the tracked value to somewhere else, creating a new data-flow; (2) indirect call instructions; (3) return instructions; (4) branch instructions. To ensure that no leaks will be missed, UNISAN always tracks each path independently and merges the tracking results (when tracking of a path returns) by calculating the union of all unsafe bytes; in other words, a byte is deemed unsafe as long as it is unsafe in one path.

Propagation to alias. Our reachability and initialization analyses are performed in a forward manner (i.e., from source to sink). Whenever an alias of the tracked value is created (e.g., by `CastInst` or `GetElementPtr` instructions), we further track the alias by merging its users into the user-graph of the current tracked value. Therefore, we do not need alias analysis for this case. However, when the tracked value is stored to another value, we need a backward slicing analysis to find the possible aliases of the store-to value, which is difficult for some cases (e.g., global variable). To handle this, we employ the simple basic alias analysis [111]. Additionally, we enforce two conservative policies to eliminate potential false negatives: (1) if we find that the aliases are pointing to a non-stack object (e.g., global variable or heap object), whose data-flow is hard to follow, we assume the tracked value is unsafe; (2) if we find the aliases is a returned value of a function call or a parameter of current function, we also assume the tracked value is unsafe.

Inline assembly. To improve performance, kernel developers commonly write inline assembly. Since inline assembly is not compiled into LLVM IR, our detection cannot be directly applied. To handle inline assembly, we manually whitelisted some safe inline assembly that will not leak the tracked value or store the tracked value to other places. All other inline assembly functions are conservatively treated as sinks.

2.1.3.2 Instrumenting Unsafe Allocations

After identifying all unsafe allocations, the initializer module of UNISAN further instruments the kernel to initialize the identified unsafe allocations by zeroing the unsafe bytes. In particular, the initialization code is inserted right after the allocation (e.g., the stack `Alloca` and `kmalloc`). Since the detection module reports unsafe allocation at the byte-level, in many cases, we do not have to initialize the whole allocated object but only the unsafe bytes. In particular, for stack allocation, we use `StoreInst` to zero the unsafe bytes if they have a continuous size of less than or equal to 8; otherwise, `memset` is used for zeroing the bytes. Heap allocation is also initialized in a similar way except that the `__GFP_ZERO` flag is passed to the heap allocator to initialize the memory if all bytes are unsafe. With the initialization, all possible uninitialized leaks cannot disclose any meaningful entropy from the kernel space, and thus can be prevented.

2.1.4 Implementation

In this section, we present the implementation details that readers may be interested in. UNISAN is built on the LLVM compiler infrastructure with version `3.7.1`. The unsafe allocation detector is implemented as two analysis module passes. One is for building the complete call graph iteratively and maintaining all necessary global context information (e.g., defined functions); the other performs the initialization and reachability analyses based on the built call graph. The unsafe allocation initializer is implemented as a transformation function pass, invoked after the detection pass. Both passes are inserted after all optimization passes. To compile the kernel into LLVM IR, we leverage the LLVMLinux project [113].

Because `llvm-link` has a symbol renaming problem, instead of merging all bitcode files into a single module, we adopt the iterative algorithm from KINT [186] to process individual bitcode files. So the input of the analysis phase is just a list of bitcode files. UNISAN is easy to use:

```
$ CC=unisan-cc make
$ unisan @bitcode.list
```

2.1.4.1 Bookkeeping of the Analysis

For each tracked allocation, we use a dedicated data structure to record its tracking results along the propagation paths. This data structure mainly includes the initialization and sinking information of each byte. The tracking history is also included to avoid repeatedly tracking a user—we do not need to track a user multiple times if the status of the tracked value is not changed.

As mentioned in Figure 2.1.3.1, UNISAN maintains the user-graph for the tracked value. The users in the graph may access different elements of the tracked value. To know which part of the tracked value is being initialized or sunk during the tracking, element information is also kept in each node of the user-graph. Moreover, we maintain *reference hierarchy* for pointers. The *reference hierarchy* is to understand if a pointer is directly or indirectly (recursively) pointing to the tracked allocation. To better understand it, let us see this example: `store i8* %A, i8** %B` stores value `A` to the memory pointed to by `B` but not `B` itself. To differentiate whether the storing operation is targeting the tracked allocation or its reference, the referencing relationship between them is required, and thus we maintain the *reference hierarchy*. Our *reference hierarchy* is straightforward: the "indirectness" is decreased by one by LoadInst; but increased by one by StoreInst. To tackle the alias problem, if StoreInst stores to non-stack memory, we assume it is sinking and stop tracking. Certainly, point-to analyses can achieve the same goal, but they are heavyweight—analyzing the kernel may take many hours.

The initialization status is updated when the tracked value is assigned another value by `StoreInst`, while the sinking status is updated when the tracked value is stored to a non-stack value in `StoreInst` or it is passed to sink functions in `CallInst`.

2.1.4.2 Tracking Different Users

Analyses are carried out by traversing the user-graph. UNISAN handles different kinds of users accordingly. After handling a user, the next users of the current user will be further tracked. In this section, we detail how the handling of each type of user is implemented.

LoadInst. `LoadInst` loads the data *pointed to* by the tracked value to the target value. We independently track both values and merge (i.e., the union operation) the unsafe bytes when both tracking processes return. Since `LoadInst` is essentially dereferencing the tracked value that is a pointer, we also increase the reference hierarchy by one in the tracking of the target value.

StoreInst. `StoreInst` stores the tracked value to the memory *pointed to* by the target value. Let us see `store i8* %A, i8** %B` again. When the tracked value is `A` (i.e., it is the value operand), we first use the conservative basic alias analysis (Figure 2.1.3.1) to find the aliases of `B`. If not all aliases can be found or some aliases are from non-stack memory, we assume `A` is sunk and update its sinking status; otherwise, we further track aliases independently and merge all unsafe bytes. Since the target value is a reference of the stored value, the reference hierarchy is decreased by one in the tracking of aliases. When the tracked value is `B`, we first consult the reference hierarchy (§2.1.4.1) to see if `B` is the tracked allocation (but not its reference); if yes, we record that the corresponding bytes of the tracked allocation are initialized.

CallInst. When the tracked value is passed to callees via argument, we recursively track the arguments in callees independently and merge all unsafe bytes. Inline assembly is conservatively handled as shown in Figure 2.1.3.1. If the called function is a sink, we record that the corresponding bytes are sunk.

GEPOperator. `GEPOperator` statements get the pointer of an element of the tracked value, which essentially creates an alias of the tracked value. The offset of the target element into the base of the tracked value and its size are calculated and maintained in the user-graph. The users of the target element are merged into the user-graph of the tracked value. The element information is the key to implement the byte-level analyses; however, when the indices of `GEPOperator` are not constants, we will not be able to obtain the element information. In this case, we stop the initialization analysis and only continue the reachability analysis, since we cannot statically decide which bytes will be initialized.

ReturnInst. `ReturnInst` is an instruction that returns a value from a function. We first use the global call-graph to find all `CallInst`s that call the current function containing the `ReturnInst`. Then these `CallInst`s are independently tracked, and unsafe bytes are merged.

CastInst, SelectInst, and PHINode. The definitions of these statements can be found at [1]. These cases are generating alias of the tracked values. We find their users and merge the users to the user-graph of the tracked value.

CmpInst, SwitchInst, BranchInst. The definitions of these statements can be found at [1]. We skip the handling for these cases, since they do not operate the tracked value.

Others. Any other cases are conservatively treated as sinks, so all uninitialized bytes in the tracked value are assumed to be unsafe.

2.1.4.3 Modeling Basic Functions

Similar to traditional static program analyses, we also modeled some basic functions. Specifically, we modeled string-related functions that typically use loops to process strings, to improve the efficiency of tracking. Moreover, we modeled some frequently used LLVM intrinsic functions (e.g., `llvm.memset`) that do not have function bodies in LLVM IR. In total, we modeled 62 simple functions by summarizing how they propagate and initialize the arguments.

2.1.4.4 Dynamic Allocations

Dynamic allocations create objects with a dynamic size. They are common (about 40%, according to our study) in heap. In general, we do not perform initialization analysis for dynamic allocations, because we cannot determine which bytes are being accessed at compiling time. As a result, we conservatively consider the whole allocation as uninitialized and only perform reachability analysis. With one exception, during the initialization analysis we will record the size value (in LLVM IR) of the dynamic allocation; if the allocation is later initialized by a basic function (e.g., `memset`) using the same size value as recorded, we consider that it is fully initialized and safe. In the instrumentation module, extra instructions are inserted to compute the size at runtime for dynamic allocations on stack, which is then passed to `memset` to initialize the memory. For dynamic allocation on heap, we utilize the `__GFP_ZERO` flag for initialization.

2.1.5 Evaluation

We systematically evaluated UNISAN to answer the following questions:

- The **accuracy** of UNISAN, i.e., to what extent can UNISAN filter out safe allocations?
- The **effectiveness** of UNISAN, i.e., whether it can prevent known and detect previously unknown uninitialized data leaks?
- The **efficiency** of UNISAN, i.e., what is the performance overhead of the secured kernel?

Experimental setup. We applied UNISAN to both x86_64 and AArch64 kernels. For x86_64, we used the latest mainline Linux (with version `4.6.0-Blurry Fish Butt`) with patches from the LLVMLinux projects [113]. x86_64 kernels were tested in a desktop machine equipped with an Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz processor and 32GB RAM. The OS is the 64-bits Ubuntu 14.04. For AArch64, we used the latest Android kernel (with version `tegra-flounder-3.10-n-preview-2`) from the Android Open Source

Table 2: Measurement results for detection accuracy of UNISAN.

Arch	Module	Static Alloca	Dyn. Alloca	Static Malloc	Dyn. Malloc	Unsafe Alloca	Unsafe Malloc	Static Bytes	Unsafe Bytes
x86_64	2,152	17,854	24	1,768	1,161	1,493	386	3,588K	863K
AArch64	2,030	15,596	32	1,790	1,233	1,485	451	11,525K	3,351K

Project, with patches from [169]. The Android kernels were tested in the Nexus 9 device that has a duo-core ARMv8 processor and 2GB RAM. We used the default configurations for both kernels.

2.1.5.1 Accuracy of Unsafe Allocation Detector

We first conducted the statistical analysis on the accuracy of the unsafe allocation detector—how many allocations are reported as unsafe. The results are shown in Table 2. In particular, there are around 2k modules (i.e., bitcode files) enabled by the default kernel configuration. It is worth noting that LLVM's optimizations aggressively inline functions and significantly opt-out most allocations—there were 156,065 functions and 413,546 static stack allocations before the optimizations. UNISAN is accurate in detecting unsafe stack allocations: only 8.4% and 9.5% stack allocations are detected as unsafe for x86_64 and AArch64, respectively. Since dynamic allocation is common for a heap object, UNISAN's detection rate is higher for heap allocations—13.2% for x86_64 and 14.9% for AArch64. UNISAN performs byte-level detection, so we also report the total number of statically allocated bytes and the detected unsafe bytes. These statistic results show that UNISAN can filter out most safe allocations to avoid unnecessary initialization.

To understand how initialization analysis and reachability analysis individually help in filtering out safe allocations, we further counted the number of unsafe allocations when we disable one of them. Specifically, in x86_64, if we disable the initialization analysis, there are 3,380 unsafe stack allocations. If we disable the reachability analysis (i.e., assuming all function calls as sinks), there are 14,094 unsafe stack allocations. In AArch64, the numbers are 2,961 and 11,209, respectively.

2.1.5.2 *Effectiveness of Preventing Leaks*

Preventing known leaks. Conservative policies (Figure 2.1.3.1) have been enforced to eliminate potential false negatives of UNISAN. To confirm that UNISAN does not miss uninitialized data leaks in practice, we selected 43 recent kernel uninitialized data leaks reported after 2013. All these leaks have been assigned with CVE identifiers. Please note that some leaks are not included because the corresponding code is not enabled by the default kernel configuration or a very similar leak has already been included. The patches of these vulnerabilities are temporarily reverted for testing. The results of the experiment are shown in Table 3. UNISAN successfully detected and prevented all these leaks without any *false negative*; hence the effectiveness in preventing existing leaks is confirmed.

Detecting previously unknown vulnerabilities. UNISAN is designed as a prevention tool that can automatically detect and fix all uninitialized data leaks at the LLVM IR level; no manual effort is required. However, to confirm that UNISAN can truly prevent new leaks and estimate the false positives due to our conservative policies, we manually review the unsafe allocations reported by the unsafe allocation detector. As shown in Table 2, UNISAN detected about 1,500 possible unsafe stack allocations and 300 possible unsafe heap allocations. Due to limited time and labor, we randomly chose 300 detected unsafe stack allocations and 50 unsafe heap allocations. In summary, we have verified 19 new uninitialized data leaks in the latest Linux kernel and Android kernel. All of these vulnerabilities are from stack, more details can be found in Table 4. All of them have been confirmed by the Linux kernel team and Android security team. Since UNISAN has not been adopted by them yet, we have to manually write the corresponding patches in source code.

2.1.5.3 *Efficiency of the Secured Kernels*

UNISAN carries out flow-sensitive, context-sensitive, and field-sensitive analyses to accurately detect the unsafe allocations, so that the performance overhead is controlled by minimizing the number of initializations. To quantify this benefit, we conducted a series of

Table 3: Tested known uninitialized data leaks. `P`: compiler padding; `M`: missing element initialization.

CVE	Mem.	Sink	Leak Bytes	Cause	UNISAN
CVE-2015-5697	heap	user	<4096	M	✓
CVE-2015-7884	stack	user	4	M	✓
CVE-2015-7885	stack	user	< 28	M	✓
CVE-2014-1444	stack	user	2	P	✓
CVE-2014-1445	stack	user	2	P	✓
CVE-2014-1446	stack	user	4	M	✓
CVE-2014-1690	stack	sock	< 16	M	✓
CVE-2014-1739	stack	user	192	M	✓
CVE-2013-4515	stack	user	10	M	✓
CVE-2013-3235	stack	user	12	M	✓
CVE-2013-3234	stack	user	12	P+M	✓
CVE-2013-3233	stack	user	12	M	✓
CVE-2013-3232	stack	user	6	P+M	✓
CVE-2013-3230	stack	user	4	M	✓
CVE-2013-3223	stack	user	1	P	✓
CVE-2013-2636	stack	sock	< 20	M	✓
CVE-2013-2636	stack	sock	4	P+M	✓
CVE-2013-2636	stack	sock	3	P	✓
CVE-2013-2636	stack	sock	18	M	✓
CVE-2013-2635	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	34	M	✓
CVE-2013-2634	stack	sock	8	M	✓
CVE-2013-2634	stack	sock	20	P+M	✓
CVE-2013-2634	stack	sock	32	M	✓
CVE-2013-2634	stack	sock	18	M	✓
CVE-2013-2547	stack	sock	< 64	M	✓
CVE-2013-2547	stack	sock	8	M	✓
CVE-2013-2237	heap	sock	1	M	✓
CVE-2013-2234	heap	sock	2	M	✓
CVE-2013-2148	stack	user	1	M	✓
CVE-2013-2141	stack	user	4	M	✓
CVE-2012-6549	stack	user	2	M	✓
CVE-2012-6548	stack	user	2	M	✓
CVE-2012-6547	stack	sock	4	M	✓
CVE-2012-6546	stack	user	2	P	✓
CVE-2012-6545	stack	sock	1	P	✓
CVE-2012-6544	stack	sock	2	M	✓
CVE-2012-6543	stack	sock	2	M	✓
CVE-2012-6541	stack	user	4	P	✓
CVE-2012-6540	stack	user	12	M	✓
CVE-2012-6539	stack	user	4	P	✓
CVE-2012-6537	stack	user	4	P	✓

extensive performance evaluations. In particular, we first used the LMBench [125] micro benchmark to evaluate the performance on core system operations (e.g., syscalls latency).

Table 4: List of new kernel uninitialized data leak vulnerabilities discovered by UNISAN. `S`: patch submitted; ✓: patch applied. `P`: compiler padding; `M`: missing element initialization; `A`: Android; `L`: Linux.

#	Sub-System	Mem.	Sink	Leak Bytes	Cause	Kernel	Patch	CVE
1	net/core	stack	sock	4	P	A&L	✓	CVE-2016-4486
2	usb	stack	user	3	P	A&L	✓	CVE-2016-4482
3	net/wireless	stack	sock	6	M	A&L	✓	AndroidID-28620350
4	net/llc	stack	user	1	P	A&L	✓	CVE-2016-4485
5	sound	stack	user	8	P	A&L	✓	CVE-2016-4569
6	sound	stack	user	8	P	A&L	✓	CVE-2016-4578
7	sound	stack	user	8	P	A&L	✓	CVE-2016-4578
8	net/x25	stack	user	8	M	A&L	✓	CVE-2016-4569
9	net/tipc	stack	sock	<60	M	A&L	✓	CVE-2016-5243
10	net/rds	stack	sock	1	M	A&L	✓	CVE-2016-5244
11	net/mac80211	stack	sock	26	M	A	S	AndroidID-28620568
12	net/wireless	stack	sock	<116	M&P	A	S	AndroidID-28619338
13	net/wireless	stack	sock	1	M	A	S	AndroidID-28620324
14	net/netfilter	stack	sock	2	M	A	S	AndroidID-28673002
15	net/netfilter	stack	sock	2	M	A	S	AndroidID-28672819
16	net/netfilter	stack	sock	1	M	A	S	AndroidID-28616963
17	media	stack	user	<192	M&P	A	S	AndroidID-28616963
18	media	stack	user	10	M&P	A	S	AndroidID-28616963
19	media	stack	user	28	M&P	A	S	AndroidID-28616963

We then used Android Benchmarks and the SPEC Benchmarks as the macro benchmarks to evaluate the performance impacts on user space programs for the protected Android kernel and Linux kernel, respectively. To measure the performance impacts on I/O intensive server programs, we further used ApacheBench to test the performance of Apache web server. All these evaluations consist of three groups: (1) native mode, in which UNISAN is not applied; (2) blind mode, in which all stack allocations and heap allocations without the `__GFP_ZERO` flag are initialized without checking whether or not they are safe; and (3) UNISAN mode, in which UNISAN is applied. In the three groups of evaluations, the kernel was replaced with the corresponding one. Note that, we did not further break down the performance overhead introduced by stack and heap, because the overall overhead is already negligible.

System Operations. In order to measure how UNISAN affects the performance of core operating system services, we used LMBench [125] as the micro benchmark. Specifically, we focus on the latency of syscalls (e.g., `null`, `write`, `open/close`, `sigaction`, etc.) and

Table 5: Evaluation results with LMBench. Time is in microsecond.

	Linux (x86_64)					Android (AArch64)				
	Native	**Blind**	(%)	**UniS.**	(%)	**Native**	**Blind**	(%)	**UniS.**	(%)
null syscall	0.04	0.04	(0.0%)	0.04	(0.0%)	0.42	0.42	(0.0%)	0.42	(0.0%)
stat	0.37	0.40	(8.1%)	0.38	(2.7%)	1.33	1.43	(7.5%)	1.37	(3.0%)
open/close	1.20	1.22	(1.7%)	1.15	(-4.2%)	6.09	6.27	(3.0%)	6.07	(-0.3%)
select TCP	2.44	2.48	(1.6%)	2.44	(0.0%)	9.67	10.67	(10.3%)	9.80	(1.3%)
signal install	0.11	0.11	(0.0%)	0.11	(0.0%)	0.58	0.58	(0.0%)	0.57	(-1.7%)
signal handle	0.58	0.71	(22.4%)	0.60	(3.4%)	1.79	1.98	(10.6%)	1.85	(3.4%)
fork+exit	156	157	(0.6%)	156	(0.0%)	851	892	(4.8%)	861	(1.2%)
fork+exec	452	464	(2.7%)	455	(0.7%)	2687	2737	(1.9%)	2742	(2.0%)
prot fault	0.295	0.317	(7.5%)	0.316	(7.1%)	1.37	1.47	(7.3%)	1.35	(-1.5%)
pipe(latency)	9.673	10.21	(5.6%)	9.909	(2.4%)	8.850	8.898	(0.5%)	8.882	(0.4%)
TCP(latency)	91.8	99.5	(8.4%)	97.3	(6.0%)	–	–	–	–	–
pipe(bandw)	3,321	3,250	(2.1%)	3,315	(0.2%)	838	728	(13.1%)	804	(4.1%)
TCP(bandw)	2,331	2,264	(2.9%)	2,333	(-0.1%)	–	–	–	–	–

impact on bandwidth (e.g., `pipe`). We ran each experiment 10 times, and the results are shown in Table 5. The last two rows are measuring bandwidth, which is the bigger the better. We do not have numbers for TCP case in Android kernel because we cannot establish TCP sockets to our Nexus 9. In the blind mode, the performance overhead could be up to 22% (the signal handle case). There are also some cases (e.g., select and pipe) where its overhead is more than 10%. Such a performance overhead is significant for the OS kernel—the foundation of computer systems. In contrast, UNISAN has a much lower performance overhead than the blind mode; its maximum overhead is 7.1% in the protection fault case. We also notice that UNISAN's performance overhead is negligible (< 1%) in many cases. On average, the performance overhead of UNISAN is less than 1.5% for both the Linux and Android kernels.

User-Space Programs. For x86_64, we used the standard SPEC CPU 2006 benchmark suite to test the performance impacts of UNISAN on user space programs. The benchmark programs were compiled with the common options (`-pie -fPIC -O2`) and each benchmark was run 10 times. The results are shown in Table 6. Specifically, the performance overhead introduced by UNISAN is only 0.54%, which is negligible. The performance overhead of the blind mode is higher than UNISAN, which is 1.92%. Note that the moderate blind-mode

Table 6: User-space (x86_64) performance evaluation results with the SPEC benchmarks. Time is in second, the smaller the better.

Programs	**Native**	**Blind**	(%)	**UNISAN**	(%)
perlbench	3.61	3.61	(0.0%)	3.59	(-0.6%)
bzip2	4.74	4.78	(0.8%)	4.78	(0.8%)
gcc	0.968	0.970	(0.2%)	0.966	(-0.2%)
mcf	2.73	2.76	(1.1%)	2.74	(0.4%)
gobmk	14.0	14.0	(0.0%)	14.0	(0.0%)
hmmer	2.07	2.03	(-1.9%)	2.04	(-1.4%)
sjeng	3.27	3.30	(0.9%)	3.33	(1.8%)
libquantum	0.0387	0.0397	(2.6%)	0.0395	(2.1%)
h264ref	9.26	9.32	(0.6%)	9.28	(0.2%)
omnetpp	0.362	0.364	(0.6%)	0.362	(0.0%)
astar	7.81	7.92	(1.4%)	7.92	(1.4%)
xalancbmk	0.0758	0.0779	(2.8%)	0.0738	(-2.6%)
milc	4.57	4.76	(4.2%)	4.76	(4.2%)
namd	8.85	8.86	(0.1%)	8.83	(-0.2%)
dealII	10.5	10.6	(1.0%)	10.6	(1.0%)
soplex	0.0166	0.0186	(12.0%)	0.0170	(2.4%)
povray	0.427	0.439	(2.8%)	0.426	(-0.2%)
lbm	1.68	1.69	(0.6%)	1.69	(0.6%)
sphinx	1.19	1.28	(7.6%)	1.20	(0.8%)
Geo-mean	(seconds)		1.92%		**0.54**%

Table 7: User-space (AArch64) performance evaluation results with multiple android benchmarks. The scores are the higher the better.

Benchmark	**Native**	**Blind**	(%)	**UNISAN**	(%)
AnTuTu	94998	92900	(2.2%)	94735	(0.3%)
Vellamo-Browser	4776	4711	(1.4%)	4729	(1.0%)
Vellamo-Metal	2766	2777	(-0.4%)	2776	(-0.4%)
Vellomo-Multicore	2610	2499	(4.3%)	2600	(0.4%)

overhead over SPEC benchmarks is when the user-space is not protected; when we also applied the blind mode to the user-space code, the overhead becomes 10% (7% from stack and 3% from heap).

For AArch64, we instead used the dedicated benchmarks for Android: AnTuTu (including 3DBench) and Vellamo. They are typical Android benchmarks to test a variety of cases (e.g., video streaming, web browsing, photo editing, etc.). Each testing was repeated five times for deriving the average numbers. The evaluation results are shown in Table 7. Again, UNISAN's performance overhead is negligible (<1%) and is lower than blind mode.

Server Programs. We used ApacheBench to test the performance impacts of UNISAN on

the latency and bandwidth of the server program—Apache web server. The ApacheBench ran in the laptop connected to our desktop machine with an one Gigabit cable. We used Apache with its default configurations. We first used a 1KB file to evaluate the latency when the concurrency (i.e., simultaneous connections) was set to be 1, 50, 100, 150, 200, and 250. Then we used files with different sizes (100B, 1KB, 10KB, 100KB, and 1MB) to measure the bandwidth (i.e., throughput) of the web server. In the bandwidth evaluation, the concurrency was set to be 32; the network I/O was not saturated except when the file size was 1MB. The request for each experiment was repeated 10,000 times. The evaluation results show that the blind mode incurs an average slowdown of 0.7% in the concurrency experiment and 0.9% in the bandwidth experiment. In contrast, UNISAN imposes an unobservable slowdown (<0.1%) in both experiments. These results confirm that UNISAN incurs almost no overhead to the server programs that are I/O intensive.

2.1.5.4 Miscellaneous

Scalability. UNISAN is scalable, which can analyze complex programs like the OS kernel. To better understand its scalability, we also measure the time for it to protect the kernels. In our physical machine, UNISAN finished the protection of kernels within 117 seconds for x86_64 and 170 seconds for AArch64.

Binary size. UNISAN inserts small initializers (e.g., zero-initialization) for unsafe allocations. To confirm that it does not significantly increase the binary size, we measured the size of the installed boot images. The results showed that UNISAN increases the size of Linux boot image from 7,417KB to 7,445KB (0.38%) and the one of Android from 6.348KB to 6.366KB (0.27%). Both are negligible.

2.1.6 Discussion

Custom heap allocator. In the current implementation, UNISAN only tracks typical heap allocators (`kmalloc` and `kmen_cache_alloc`). To handle custom heap allocations (e.g., `alloc_skb`), UNISAN can use the specifications of allocators provided by developers. In

fact, for user space programs, LLVM already provides an API `isMallocLikeFn` to test whether a value is a call to a function that allocates uninitialized memory based on heuristics. We plan to also use heuristics to infer "malloc-like" functions in OS kernels.

Source code requirement. Source code is required. In case that some kernel drivers are close-sourced, we have to carefully identify all possible calls that target these drivers and assume all such calls as sinks. Failures in identifying such calls will result in incompleteness of call-graph and thus false negatives.

Security impacts of zero-initialization. Some systems use uninitialized memory as the source of randomness. For example, the `SSLeay` implementation of `OpenSSL` uses uninitialized buffer as entropy source to generate random number (see `ssleay_rand_bytes()`). Zero-initialization will clearly reduce the entropy of such a source of randomness; however, we argue that using such a source of randomness is insecure and should be avoided—reading uninitialized data is classified as memory error.

False positives. In many cases, UNISAN eliminates false negatives by sacrificing accuracy (i.e., increasing the false positive rate). There is still room to reduce the false positives. For example, point-to analysis [76] can help find indirect call targets, and dynamic taint analysis [154] can help handle cases like inline assembly. Considering that false positives do not affect program semantics but just introduce more performance overhead and that UNISAN is already efficient, we leave these optimizations for future work.

More kernel modules. In our experiments, we included only modules enabled by the default kernel configurations. Since the current Linux kernel has around 20,000 modules total, a majority of them have not been included in our evaluation. Unfortunately, due to some GCC-specific features, some modules are still not compilable by LLVM and require extra engineering effort to patch. Since UNISAN is a proof-of-concept research project, we believe supporting these additional modules is out-of-scope. We may be able to rely on the open source community to provide the required patches (e.g., the LLVMLinux project [113]) or port UNISAN to GCC.

Beyond kernels. UNISAN's detection and instrumentation work on the LLVM IR level, and thus it can be naturally extended to protect user space programs. Specifically, to support user space program, IR of libraries should also be included, and sources (for heap) and sinks should be re-defined. As a future work, we will use UNISAN to detect and prevent information leaks in security- and privacy- sensitive programs (e.g., OpenSSL).

2.1.7 Related work

Kernel leak detection and prevention. The most related works are Peiró's model checking based kernel stack leak detection [148], PaX STACKLEAK plugin [175], and Split kernel [96].

Peiró's model checking is essentially a simple taint tracking from stack allocation to `copy_to_user`. If there is no assignment or `memset` between the allocation and `copy_to_user`, a leak is reported. Such an approach has some limitations: it does not track targets of indirect calls, different sinks, the propagation of uninitialized data, or handle partial initialization, so many leak cases will be missed. In UNISAN, we accurately track each byte of an allocation and eliminate false negatives thoroughly.

`STACKLEAK` simply clears the used kernel stack when the control is transferred back to the user space. Such an approach cannot prevent leaks that disclose data generated in the current syscall. Split kernel [96] instead clears the stack frame whenever a function is called. Both approaches impose significant performance overhead (see Table 5) because they need to frequently zero-out blocks of memory. Compared with UniSan, these approaches provide a broader security, as they also prevent other uninitialized data uses (e.g., uninitialized pointer dereferencing).

None of above approaches can protect heap allocations. To the best of our knowledge, UNISAN is the first approach that prevents both stack and heap uninitialized data leaks efficiently.

Detecting uninitialized memory accesses. Traditional uninitialized data read detection

and undefined behavior [185] detection's can also detect uninitialized data leaks.

Both LLVM and gcc provide the `-Wuninitialized` option to detect uninitialized data use. First, they only perform intra-procedure analysis; however, all the uninitialized data leaks propagate data across function boundary (e.g., calling `copy_to_user`); hence they cannot detect such leaks. Second, their analysis does not handle many common cases. For example, reading the uninitialized data through its pointer cannot be caught.

Some dynamic tracking techniques [25, 138, 162] have been proposed to detect uninitialized data reads. These techniques rely on dynamic instrumentation platforms (e.g., Valgrind [134] and DynamoRIO [25]) to instrument the binary and analyze memory access patterns dynamically. These tools can report uninitialized data use without false positives; however, their performance overhead (>10x) is too high for them to be adopted as prevention tools.

Differently, MemorySanitizer [172] relies on compile time instrumentation and shadow memory to detect uninitialized data use at run-time. Its performance is much better than dynamic instrumentation based tools; however, it still imposes a 3-4x performance overhead. Usher [191] proposes value-flow analysis to reduce the number of tracked allocations to reduce the performance overhead of MemorySanitizer to 2-2.5x.

Memory safety and model checking. Many memory safety techniques [129, 131, 132, 160] have been proposed to prevent spacial memory errors (e.g., out-of-bound read) and use-after-free bugs, and model checking techniques [13, 126] can detect semantic errors caused by developers. These tools are effective to detect or prevent kernel leaks caused by spacial memory errors, use-after-free, or semantic errors, and thus are orthogonal to UNISAN. DieHard [16] probabilistically detects uninitialized memory uses for heap but not stack. Cling [4] constrains memory allocation to allow address space reuse only among objects of the same type, which can mitigate exploits of temporal memory errors (e.g., use-after-free and uninitialized uses). StackArmor [37] is a sophisticated stack protection that also prevents uninitialized reads in binaries. Since StackArmor uses stack layout randomization to prevent

inter-procedural uninitialized reads, such a protection is probabilistic. Type systems [58, 74] can prevent dangling pointer and uninitialized pointer dereferencing, and out-of-bound accesses; however uninitialized data leaks are not covered.

Protections using zero-initialization. Zero-initialization has been leveraged to achieve protections in previous works. Secure deallocation [39] zero-initializes the deallocated memory to reduce the lifetime of data, thus reducing the risk of data exposure. Lacuna [61] allows users to run programs in "private sessions". After a session is over, all memory of its execution is erased. In general, zeroing-upon-deallocation has two issues: (1) deallocations are not always available (e.g., deallocation is missing in the case of memory leak); (2) it is hard to selectively zero deallocations for efficiency, as discussed in §2.1.2.2. StackArmor [37] only zero-initializes intra-procedural allocations (i.e., not be passed to other functions) that cannot be proven to be secure against uninitialized reads.

2.2 POINTSAN*: Eliminating Uninitialized Pointer Uses*

In programming languages such as C and C++, programmers decide whether to initialize a variable with a deterministic value when it is allocated. C enthusiasts often argue that if programmers know that the code will later set a proper value anyway, initialization on allocations is an unnecessary use of precious CPU cycles. This argument makes sense from a functional point of view since such an unnecessary use of CPU cycles can cause a significant runtime overhead if it occurs up to millions of times per second, as it does in programs such as in OS kernels. However, manually keeping track of all possible code paths to ensure proper initialization is an error-prone task. Even worse, automatic detection of uninitialized use, such as the warning of compilers, is inaccurate for several reasons. First, inter-procedural tracking often leads to false positives and false negatives because of problems such as aliasing. Second, whether an uninitialized-use warning is justified is highly subjective: While some programmers may prefer a warning in every possible case, others might consider a warning unnecessary if it would not cause an observable error or is

likely a false positive.

Uninitialized data represents arbitrary values that were coincidentally stored in the memory. If the uninitialized data is used for a memory read, such as the case in which an uninitialized data pointer is dereferenced, the whole memory region can be read, causing information leaks. Despite their potentially dangerous consequences, uninitialized-use bugs are very seldom classified as security vulnerabilities [67, 185], which arguably originates from the perception that it is hard for an attacker to control the memory layout in order to make dereferencing exploitable. In particular, widely used systems such as OpenSSL explicitly use uninitialized data for the generation of entropy (see function `ssleay_rand_bytes()` in the `SSLeay` implementation) and hence ground their security on the assumption that such data is impossible to control or predict. On the other hand, our study revealed that in 2015 and 2016 alone, although 16 uninitialized-use vulnerabilities have been patched in the Linux Kernel, only *one* was reported for a Common Vulnerabilities and Exposures ID (CVE). In fact, since 2004, only eight uninitialized-use vulnerabilities in the Linux kernel have been reported for a CVE. From a security point of view, uninitialized use or more precisely, temporal memory errors should be included as a prevention target in state-of-the-art memory protection mechanisms. However, advanced security mechanisms such as SoftBound+CETS [131, 132] and WatchdogLite [129]), which claim full memory safety, do not currently cover uninitialized uses.

In this work, we first show that uninitialized pointers can be reliably controlled by attackers without the need of an additional memory-corruption vulnerability or special assumptions. Therefore, an uninitialized-pointer use vulnerabilities can be reliably exploited to achieve arbitrary memory read (i.e., causing information leaks), write, and code execution. In particular, we show that almost the whole kernel stack is controllable to a *local attacker* by either executing syscalls based on how they leave the stack after they return or exhausting memory and guiding stack allocation. We first survey existing reported and patched uninitialized-use vulnerabilities in the Linux kernel and then propose the reliable targeted

stack-spraying technique to write and retain arbitrary data on the kernel stack. The core of the fully automated targeted stack spraying includes a deterministic stack spraying technique and a reliable exhaustive memory spraying technique. The deterministic stack spraying technique consists of three components: a *tailored symbolic execution engine* that explores paths and outputs the concrete parameters to trigger the paths; a *guided fuzzer* that takes as input information generated by the symbolic execution engine to verify that stack control is indeed achieved; and a *coordinator* that safely and efficiently parallelizes the symbolic execution engine and the fuzzer. The exhausting memory spraying technique complements deterministic stack spraying by strategically consuming a huge region of memory to guide stack allocations and preparing malicious data in the memory pages that will be used by the guided stack allocations. Combining both approaches allows us to reliably control almost the whole kernel stack.

We further propose POINTSAN, a mitigation that defeats uninitialized-use exploits. POINTSAN is inspired by the observation that uninitialized-use exploits usually control an uninitialized pointer to achieve arbitrary read/write/execution. By zero-initializing pointer-type fields in an allocated object, we can prevent an adversary from controlling these pointers. Since memory page at the address zero is not accessable in Linux, zero-initialization becomes a safe prevention operation. More specifically, we perform an intra-procedural analysis for the Linux kernel source code. We realize both the analysis that identifies unsafe pointer-type fields and the instrumentation that zero-initializes the identified pointer-type fields based on the LLVM compiler Infrastructure. We evaluated the performance and relibiblity of POINTSAN using LMBench and the SPEC benchmarks. Both the LMBench and SPEC benchmark results confirm that POINTSAN is very efficient and reliable (no single error was observed during the evaluation).

2.2.1 Uninitialized Uses and the Kernel Stack

2.2.1.1 Uninitialized Uses in OS Kernels

In this section, we present uninitialized-use issues in OS kernels. We first investigate how widespread uninitialized-use vulnerabilities actually are in the Linux kernel and how aware people are of this problem. To this end, we have manually analyzed the reported CVE entries that lead to privilege escalation attacks in the Linux kernel since 2004 [57], and the commit log of the Linux kernel git repository [180], which dates back to 2005. To reduce the huge number of commits to a manageable size, we mostly concentrated on the commit log messages between the years 2015 and 2016. For the CVEs, we find that eight out of 199 (4%) privilege escalation vulnerabilities reported since 2004 are caused by the use of uninitialized objects or pointers. For Linux kernel commit messages, we first identified candidates of uninitialized use by inspecting the commit messages using keywords such as *uninitialized pointer dereference* and *undefined pointer*, which resulted in 52 candidate commits from 2015 and 2016, 28 of which were subsequently filtered out by our manual analysis because they are not exploitable (e.g., NULL pointer dereference bugs). Out of the remaining 24 cases, eight are uninitialized pointer-based reads, which can lead to information leaks, and 16 are uninitialized pointer-based writes or function calls. We further inspected these 16 write/call cases and found that 11 cases (69%) are from the stack while only five cases are from the heap. These findings not only show that uninitialized-use vulnerabilities are quite common in the Linux kernel but also indicate that these vulnerabilities are not considered particularly security-relevant or even not reported at all. Moreover, our findings confirm that most uninitialized variables are from the stack rather than the heap, which is a significant difference from use-after-free vulnerabilities.

2.2.1.2 Kernel Stack Management

Since most uninitialized variables are from stack, our primary focus lies on vulnerabilities caused by uninitialized uses of stack variables and pointers in the Linux kernel, and thus

understanding how Linux manages its kernel stacks and which features it offers in this regard is important. In Linux, every thread has its own kernel stack allocated in the kernel memory space with the maximum size of the stack depending on the specific Linux version. In general, the stack is 4KB or 8KB for a 32-bit OS (x86) and 8KB or 16KB for a 64-bit OS (x86-64), which is quite small compared to the default stack size soft limit of 8MB for Linux user space stacks. The special data structure `struct thread_info`, whose size is 104-byte in our system, is saved at the stack top (low address). The fundamental goal behind limiting the kernel stack size is to limit overall memory consumption when a large number of threads is running in the kernel in parallel, and each thread has its own kernel stack. Because of the limited stack size, storing large variables on the kernel stack and creating long call chains in the kernel space is discouraged. To ensure that the stack depth is shallow enough to avoid a stack overflow, Linux provides the `checkstack.pl` tool for static analysis of the stack. Although the small size of the Linux kernel stack improves the success rate of a stack-spraying attack, the shallow stack depth (or the lack of loops and recursions) limits the spraying range. Besides normal thread stacks, Linux also has other specialized stack types. For example, while debug stacks are used for hardware (interrupt1) and software (INT3) debug interrupts, interrupt stacks are used for external hardware interrupts or for processing software interrupts. Since these stacks do not accept user-controlled data, we do not take them into account and instead focus on normal per-thread kernel stacks that are used when syscalls are issued.

2.2.1.3 Stack Usage of Syscalls

The more frequently a stack region is used, the more likely an uninitialized variable will reside in this region. Therefore, taking control over frequently used memory regions increases the success rate of an uninitialized-use attack. We hence analyze stack usage of syscalls to understand which portions of the kernel stack are most frequently used.

To profile stack usage of syscalls, we use `kprobes` to intercept syscall enters and returns,

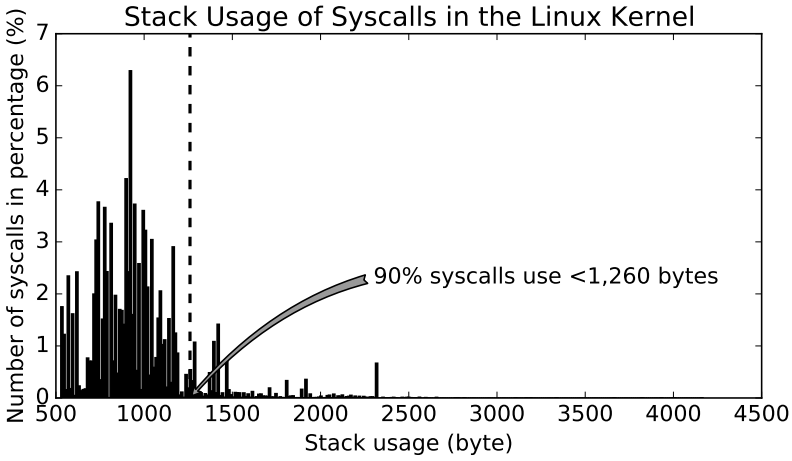

Figure 9: The profile for stack usage of syscalls in the Linux kernel. The total size of the kernel stack is 16KB. 90% syscalls use less than 1,260 bytes aligned to the stack base. The average stack usage is less than 1,000 bytes.

and scan the stack memory to check maximum stack usage of these syscalls. Specifically, upon syscall enter, we zero out the stack memory and continue the normal execution; upon syscall return, we scan the stack memory from stack top (i.e., the lowest stack address) until we find the first non-zero byte. We conservatively treat the offset of the first non-zero byte into stack base (i.e., the value of stack pointer upon syscall entry) as the maximum stack usage of the syscall. We use the `Trinity` fuzzer to invoke all syscalls to obtain stack usage for all syscalls. Because `Trinity` usually takes a long time to explore a syscall or even just does not terminate, we set five-second timeout for fuzzing each syscall. Figure 9 summarizes the maximum stack usage for all syscalls. In particular, we find that (1) the average stack usage of syscalls is less than 1,000 bytes (aligned to the stack base at high address) and (2) 90% syscalls use only the highest 1,260 bytes on the stack. It is important to note that the stack usage represents the maximum stack region a syscall uses. Assuming stack objects are uniformly distributed in stack regions used by syscalls, we find that the average location of stack objects is 510 bytes into the stack base and more than 90% stack objects are allocated in the highest 960-byte stack region. Therefore, *the highest 1KB stack region is frequently used and thus is the primary target of our spraying*.

2.2.2 The Exploitability of Uninitialized Uses

The main challenge in exploiting uninitialized uses is to control data in the uninitialized memory. By planting malicious pointers in the target memory, an uninitialized pointer dereference can be turned into arbitrary memory read—causing information leaks, write, or code execution. However, unlike heap spraying, in which the number and the size of allocated heap objects are user-controlled, stack spraying has the additional problem of stack objects usually being static and fixed in size. The placement of the Linux `thread_info` structure, at the stack top, requires the stack size to be limited; otherwise, stack buffer overflows may occur. In addition, kernel space is shared by all threads. Not limiting the size of stack will easily exhaust memory. Therefore, Linux kernel developers are encouraged to use the script (`scripts/checkstack.pl`) to statically analyze stack usage. The script in particular checks the stack usage (in bytes) of each function so that developers can find functions that use too much stack memory. Because of these features—the limited stack size, the static and fixed-size stack objects, and the stack usage check, a targeted stack-spraying attack is significantly more difficult than a heap-spraying attack.

To enable a targeted stack-spraying attack in the kernel space, we need to prepare malicious data in a specific location of the kernel stack in the presence of aforementioned difficulties. Specifically, the location itself needs to be chosen in such a way that the uninitialized memory will overlap the prepared data. In general, we can store malicious data in such a location in two ways: (1) finding an execution path that prepares data overlapping that of the vulnerability and (2) finding a way to guide the kernel to allocate stacks on the memory pages with prepared data. The first method is deterministic: Once such a path and its triggering inputs are found, we can deterministically write arbitrary data at the specified location. Since the data is saved at the target location by normal execution paths, this method is stealthy and hard to detect. By contrast, the second method affects the stack allocation of another process/thread by exhausting memory, which can be reliable but not fully deterministic. This method can achieve broad control because the overlapping is at page

level. However, since the creation of a new process/thread executes kernel functions that use the kernel stack, a portion (near the stack base) of the prepared data will be overwritten. As a result, the second method loses control of the stack region at high address. As mentioned in §2.2.1.3, our primary spraying target is the highest 1KB stack region. To control this region, we have to use the first method. For these reasons, we combine both methods so that attackers can achieve reliable or even deterministic control over a broad stack region. In this section, we present an overview of both methods.

2.2.2.1 Deterministic Stack Spraying

We design the deterministic stack spraying technique, which finds an execution path that prepares data overlapping that of an uninitialized variable. The main challenge of deterministic stack spraying is to find a syscall with specific parameters that will trigger an overlapping execution path. An overview of the technique used for the attack is shown in Figure 10. The technique consists of three components: a symbolic execution engine, a guided fuzzer, and a coordinator that handles communication between the symbolic execution engine and the guided fuzzer. The goal of the symbolic execution engine is to explore as many execution paths as possible to find one that saves user-controlled data on that stack, which will overlap an uninitialized variable. However, symbolic execution is prone to the path explosion problem because of that the number of feasible paths in a program can be infinite when the program contains unbounded loop iterations. A possible solution for this problem is to use heuristics for either path-finding or concretizing the loop condition. To achieve high coverage in path exploration, we follow the second method: During symbolic execution, we concretize the loop conditions and at the same time, identify loops and their symbolic conditions, and then let the fuzzer selectively explore these loops. To verify whether a syscall can actually save arbitrary data on the kernel stack, our guided fuzzer replaces the non-controlling parameters (that are confirmed not to affect execution paths during symbolic execution) with a magic code. When the syscall returns, we use `kprobes` to intercept the

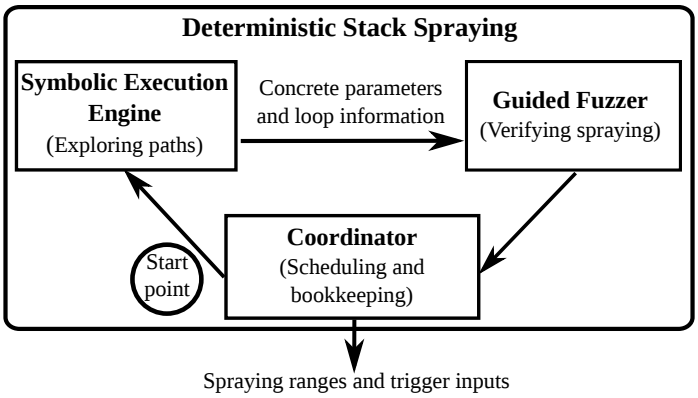

Figure 10: Overview of the architecture of our deterministic stack spraying technique, which automatically analyzes all syscalls and outputs results, including which range of the stack we can control and how to control it.

execution and scan the kernel stack to check which ranges of the stack memory have been polluted by magic code. These ranges are those we can control.

2.2.2.2 Exhaustive Memory Spraying

The exhaustive memory spraying technique guides the stack allocation of a new process or thread so that the memory pages used by the stack overlap those with prepared data. The main challenge of such a technique is to improve the reliability of the overlapping. To overcome this challenge, we design a strategy that reliably controls the memory of the kernel stack. Specifically, our exhaustive memory spraying technique includes two steps: (1) occupying the majority of memory in the target machine and (2) polluting all the available remaining memory with malicious data (for uninitialized variables). Memory occupation forces the kernel to use the remaining memory, which is small, for the newly allocated kernel stacks. Because the remaining memory is small, the pollution operation can be done quickly and effectively. Once we ensure that almost all available memory is polluted by malicious data, the memory of the newly allocated stacks will contain the malicious data. Note that in the kernel space, the kernel does not zero out the allocated memory pages, so the malicious data will not be cleared.

2.2.2.3 *Evaluation of Exploitability*

We have implemented targeted stack-spraying and evaluated its effectiveness with regard to exploiting uninitialized-use vulnerabilities by measuring the control coverage we achieved. We present the total stack ranges that we can control with deterministic stack spraying and exhaustive memory spraying, the distribution of controlled regions, and the time spraying takes. In particular, we investigate the following questions:

- **Stack spraying coverage.** What is the overall range of the kernel stack can we control with our two spraying techniques?
- **Coverage distribution and frequency.** In deterministic stack spraying, how is the control coverage distributed over the kernel stack? And how frequently can we control a specific stack region?
- **Spraying reliability.** In exhaustive memory spraying, how reliably can we control memory?
- **Spraying efficiency.** How long does it take for our spraying techniques to achieve memory control?

Experimental Setup. We obtained the symbolic execution engine S2E from the master branch of its git repository[1], which uses QEMU 1.0.50 and clang 3.2. Our guided fuzzer is based on Trinity version 1.7pre. Both the symbolic execution and guided fuzzer run on virtual machines with Debian 8.5.0 (64-bit) on Linux kernel version 3.16. Syscalls that do not have parameters (e.g., `getpid`) and therefore cannot take user-controlled data are ignored. Also, non-x86/x86_64 syscalls are ignored. Out of 313 syscalls available in the kernel source, we selected 229 for the evaluation. The stack of the Debian system is 16K-byte. The stack has two regions that are at fixed locations and cannot be sprayed:

[1] https://github.com/dslab-epfl/s2e.git as of August 2016

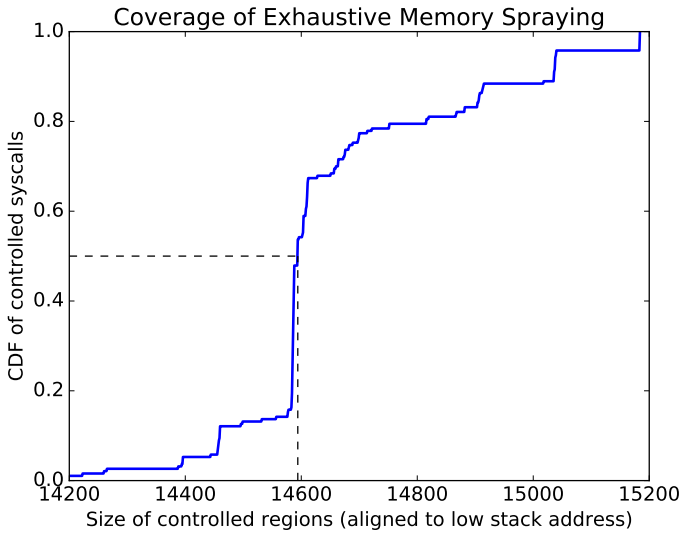

Figure 11: The cumulative distribution (CDF) of coverage achieved by exhaustive memory spraying. Its average control rate is about 90%. The memory (1,700 bytes on average) near the stack base cannot be controlled.

the lowest 104 bytes reserved for `thread_info` and the highest 160 bytes reserved for OS operations such as context switches. In all evaluations, the magic code is set to be 4-byte string "UBIE".

Stack Spraying Coverage. We evaluated the coverage for deterministic stack spraying and exhaustive memory spraying separately and then measured their combined coverage. In both scenarios, we used 229 pre-selected syscalls for the evaluation.

In deterministic stack spraying, we found that only 34 syscalls do not allow us to take control of any stack region. After manual inspection, we concluded that this is because these syscalls do not admit any parameters that will be stored on the stack. Table 8 summarizes the amount of bytes that can be controlled by the top 10 syscalls. In the highest 1KB stack region, which is frequently used (§2.2.1.3), deterministic stack spraying covers 315 bytes using all available syscalls. Hence, 32% of the frequently used region of the kernel stack can be manipulated using deterministic stack spraying.

Stack memory after the highest 1KB is subjected to exhaustive memory spraying. As mentioned in §2.2.2, a portion of the prepared malicious data in the kernel stack of a victim process by exhaustive memory spraying is likely to be overwritten because of some kernel

operations (e.g., setting up a new process) in the victim process. To evaluate which areas are overwritten, we enabled exhaustive memory spraying and ran the `Trinity` fuzzer to invoke syscalls. We then used `kprobes` to intercept syscall entry points and check which regions have been polluted by magic code (indicating that they were successfully sprayed). Figure 11 shows the results: Besides a small overwritten region near stack base, the remaining region can be fully controlled. The size of the uncontrollable region varies. On average, the highest 1,722 bytes at the stack base are overwritten, and in some cases, this region can be as small as 1,200 bytes. Overall, while losing control of this region, exhaustive memory spraying retains control of all other stack memory, achieving an average coverage rate of 89%.

Deterministic stack spraying and exhaustive memory spraying work as two complementary techniques: While exhaustive memory spraying retains the majority of the memory, it cannot control the frequently used stack region. Deterministic stack spraying complements it by controlling 32% memory of the frequently used stack region. Overall, by combining both spraying techniques, targeted stack-spraying reliably controls more than 91% of the kernel stack.

Coverage Distribution and Frequency. We further investigated how the control coverage is distributed over the kernel stack when using deterministic stack spraying. Figure 12 presents the distribution. We found that the coverage ranges from 200 to 800 bytes. More importantly, the control with deterministic stack spraying is highly concentrated: Some regions can be controlled by more than 100 syscalls. We believe these regions are most likely used by stack objects, and uninitialized variables likely reside in these regions, so controlling these regions is critical to exploit uninitialized uses from the kernel stack. Table 8 presents top 10 syscalls with high coverage. Syscalls `vmsplice`, `uname`, and `fcntl` have the highest individual coverage. We further investigated which regions of the stack are uniquely controlled by a syscall. Table 9 contains all syscalls that uniquely control a region. Overall, only 80 bytes are uniquely controllable by a single syscall. Other covered bytes can be controlled with multiple syscalls, thus their sprayers are more reliable.

Table 8: Top 10 syscalls with the highest control coverage in the kernel stack.

System call	Coverage (Byte)
vmsplice	224
uname	99
fcntl	96
setpriority	88
sched_get_priority_min	88
sched_get_priority_max	88
personality	84
iopl	84
umask	80
io_destroy	76

Table 9: Syscalls that uniquely control a stack region. Unique coverage is represented by the number of uniquely controlled bytes.

System call	Unique Coverage
wait4	16
waitid	12
timerfd_create	8
clock_getres	8
fcntl	8
mq_open	8
sched_rr_get_interval	8
mq_notify	8
timer_gettime	4
Total	**80**

Reliability of Exhaustive Memory Spraying. We investigated the reliability of exhaustive memory spraying by measuring how likely the kernel uses the sprayed memory for the kernel stack, i.e. whether the allocated stack memory overlaps the one with prepared data. Specifically, we enable the exhaustive memory spraying and run the `Trinity` fuzzer to invoke syscalls. Then we count the number of times (i.e., probes) a syscall has been invoked until we find that the kernel stack for the syscall is sprayed. After running all syscalls with `Trinity`, we found that in most cases, the kernel uses the sprayed memory as stack in the first or second probe. The average number of probes we achieve overlapping is 1.8. The worse case is less than 10 probes. Such results show that the exhaustive memory spraying technique is very effective and thus can reliably control the uninitialized memory.

Efficiency of Spraying. In deterministic stack spraying, both the symbolic execution and

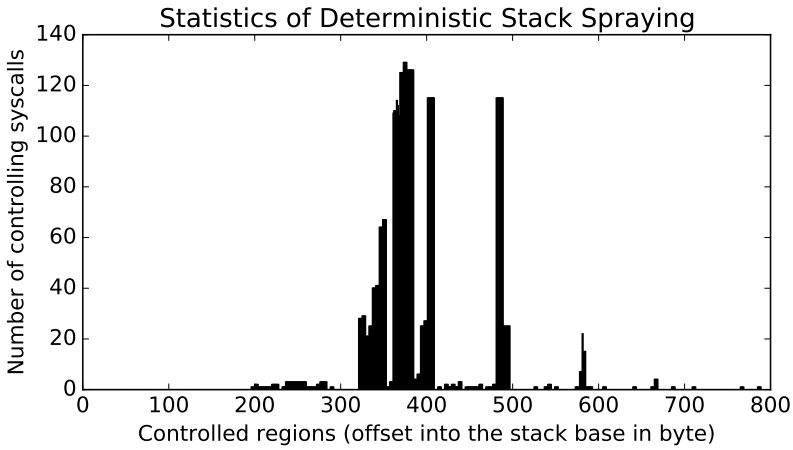

Figure 12: The coverage, distribution, and frequency of stack control achieved by the deterministic stack spraying technique.

the guided fuzzing are time-consuming processes. In many cases, they do not terminate even after running for a few hours. To handle this problem, we have set a timeout for this analysis: If the analysis for a syscall does not generate new coverage within a pre-set timeout, we forcibly terminate the analysis for this syscall and continue to analyze next one. After experimenting with various timeouts, we ultimately set the timeout to 30 minutes. We found that the vast majority of syscalls can be thoroughly analyzed within this time-frame, with only 12 syscalls not finishing in time. To improve the spraying efficiency, we empirically prioritize three types of syscalls: (1) syscalls that set configurations or write data. Such syscalls (e.g., `pwritev`) are very likely to save data on kernel stack; (2) syscalls that contain loops. Such syscalls usually affect a large region of theÂůkernel stack; and (3) syscalls that contain (multiple) pointer-typeÂůparameters. With such a syscall ranking, we were able to control more than 200 bytes in the frequently used region within a few minutes. Compared to deterministic stack spraying, exhaustive memory spraying is much more efficient. The time memory occupying takes depends on the size of the available memory. In our case, the memory is 512MB, and the time for occupying the memory is less than 2 seconds. Since memory pollution writes data into a small memory region, its time is unobservable.

Case Study. The targeted stack-spraying technique provides a conceptual approach for exploiting a given uninitialized-use vulnerabilities by preparing malicious data at a target

stack location. For the sake of illustration, we exemplify the applicability of our approach by adapting Cook's exploit [41].

To the best of our knowledge, Cook's exploit is the only one that exploited an uninitialized-use vulnerability (CVE-2010-2963) in the Linux kernel stack. Figure 13 shows how the code is subject to the uninitialized use. The pointer `data` in object `karg.vc` is not initialized but dereferenced in function `copy_from_user()`. Cook exploited this vulnerability by tuning the `cmd` argument to let the union struct adopt the type of `struct video_tuner`, causing `karg.vt` to be written with user-controlled data. For such a spraying attack to succeed, at least four requirements must be satisfied: (1) The object having the uninitialized pointer must be contained by a union struct; (2) another type in the union struct has to have a non-pointer field that overlaps with the uninitialized pointer because users are not allowed to specify pointers pointing to kernel space; 3) this non-pointer field can be overwritten with user-controlled data; and 4) the user-controlled data will not be cleared. An execution path satisfying all these requirements is uncommon in practice, and finding such a path manually is unrealistic in most cases.

In this case study, we show that our targeted stack-spraying technique can automatically find many execution paths that are able to prepare a malicious pointer on the kernel stack, thus controlling the uninitialized pointer `kp->data`. To reproduce Cook's exploit, we installed version `2.6.27.58` of the Linux kernel in 64-bit Ubuntu 14.04; the kernel source code in file `compat_ioctl32.c` was reverted to contain the vulnerability described in CVE-2010-2963. Determined by the operating system, the size of the kernel stack is 8KB instead of 16KB in this case study. To benefit from our targeted stack-spraying technique, we need to find out the location of the uninitialized pointer in the stack. To get the pointer location, we used `kprobes` to hook the function `do_video_ioctl`. The handler provided by `kprobes` enables us to find the location of the stack pointer when `do_video_ioctl` is called. Using this information, we computed the offset of the uninitialized pointer `kp->data` from the stack base, which is `396`. After knowing this offset, we employed our deterministic

```
static long do_video_ioctl(struct file *file, unsigned int cmd,
        unsigned long arg) {
    union {
        struct video_tuner vt;
        struct video_code vc;
    } karg;
    ...
    /* karg.vc contains an uninitialized pointer */
    err = get_microcode32(&karg.vc, up);
    ...
}
int get_microcode32(struct video_code *kp,
        struct video_code32 __user *up) {
    ...
    /* uninitialized pointer is dereferenced */
    copy_from_user(kp->data, up->data, up->datasize))
    ...
}
```

Figure 13: The uninitialized-use vulnerability used in Cook's exploit.

stack spraying technique to find syscalls that can prepare a 8-byte malicious pointer at this offset. Altogether, we were able to find 27 such syscalls with corresponding parameters. Independent of the chosen syscall, we could prepare a malicious pointer at the target offset, resulting in an arbitrary write.

This case study shows how to use the proposed deterministic stack spraying technique to find syscalls that can control a specific location on stack. It also confirms that control of the stack can be achieved generally and automatically, and, in the presence of a suitable uninitialized-use vulnerability, a successful exploit can be built reliably and readily.

2.2.3 Defeating Uninitialized-Use Exploits

We showed that uninitialized-use vulnerabilities can be readily and reliably exploited using our targeted stack-spraying technique. While use-after-free and buffer overflow problems have been extensively studied, which has resulted in various protection techniques (e.g., memory safety), the uninitialized-use problem has rarely received attention. Our findings show that uninitialized use constitutes a severe attack vector that calls for practical defense mechanisms; however, to date no practical defense mechanisms exist. As such, we designed and implemented an efficient and practical mitigation that counters uninitialized uses. Our mitigation is inspired by the observation that uninitialized-use exploits usually control an

uninitialized pointer to achieve arbitrary read/write/execution. By zero-initializing pointer-type fields in an allocated object, we can prevent an adversary from controlling these pointers. Since memory page at the address zero is not accessable in Linux[2], zero-initialization becomes a safe prevention operation. More specifically, we perform an intra-procedural analysis for the Linux kernel source code. We realize both the analysis that identifies unsafe pointer-type fields and the instrumentation that zero-initializes the identified pointer-type fields based on the LLVM compiler Infrastructure [112].

2.2.3.1 Identifying Unsafe Pointer-Type Fields

Our analysis is carried out on the LLVM IR, so type information is preserved. In most cases, we can differentiate pointer-type fields from other fields based on type information. We start our analysis from allocation sites (i.e., `AllocaInst` instructions in LLVM IR). The first step is to identify all pointer-type fields by recursively (a field could be a struct or an array type) traversing each field in the allocated object. Since integer values might be used as a pointer, we also treat the 8-byte-integer type as a pointer type.

To initialize identified pointer-type fields, we could conservatively zero out them upon allocations. This strategy, however, will overly initialize many already initialized pointers and therefore introduce unnecessary performance overhead. To reduce initialization overhead while still ensuring security, we designed an intra-procedural program analysis that checks the following two conditions: (1) the pointer field is not properly initialized in the function; and (2) the pointer is sinking (e.g., being used or passed to other functions). Only those pointer-type fields satisfying both conditions require zero-initialization. More specifically, once all pointer-type fields are identified, we perform taint analysis to keep track of the initialization status and sinking status of the pointer-type fields in the following conservative ways:

[2]Since the Linux kernel with version 2.6.23, the `/proc/sys/vm/mmap_min_addr` tunable was introduced to prevent unprivileged users from creating new memory mappings below the minimum address

- When a pointer-type field is explicitly assigned by other values (i.e., it is the store-to target in a memory storing instruction (`StoreInst`)), we record that this field is initialized.

- When a pointer-type field that is not fully initialized is passed to other functions as a parameter or stored to memory, we report it as unsafe, which thus requires initialization.

- When a pointer-type field that is not fully initialized is dereferenced (i.e., used as the pointer argument in memory loading instruction (`LoadInst`), `StoreInst`, or function call instruction (`CallInst`), we treat it as unsafe as well.

The basic alias analysis [111] provided by LLVM is adopted to tackle the alias problem, so accessing pointer-type fields via their aliases is also tracked. Since our analysis is intra-procedural, such a basic alias analysis suffices for the purpose of efficiently detecting pointer-type fields that lack proper initialization. With this conservative taint analysis, we managed to reduce the number of to-be-initialized bytes from 105,960 to 66,846.

2.2.3.2 Implementation

Both the analysis pass and the instrumentation pass are implemented with LLVM. Both passes are inserted after all optimization passes. To use the mitigation, users only need to specify the option (i.e., `-fsanitize=init-pointer`) when compiling the kernel source code. To compile Linux kernel source code with LLVM, we applied the patches provided by the LLVMLinux project [113]. The zero-initialization code is inserted right after allocation sites. In LLVM IR, inline assembly is invoked by a `CallInst`, which is treated as a sink in our analysis, so the common inline assembly in Linux kernel is not an issue.

2.2.3.3 Evaluating Pointer Initialization

To confirm that our mitigation is practical, we applied it to the latest Linux vanilla kernel (x86_64, version `4.7`) and evaluated its performance. The testing is performed in the virtual

Table 10: LMBench results. Time is in microsecond.

System call	Baseline	W/ defense	Overhead(%)
null syscall	0.04	0.04	(0.0%)
stat	0.42	0.40	(-4.8%)
open/close	1.20	1.14	(-5.0%)
select TCP	2.44	2.62	(7.4%)
signal install	0.11	0.11	(0.0%)
signal handle	0.60	0.64	(6.7%)
fork+exit	163	157	(-3.7%)
fork+exec	447	460	(2.9%)
prot fault	0.327	0.356	(8.9%)
pipe	8.906	9.058	(1.7%)
TCP	25.6	27.5	(7.4%)
Average			**1.95%**

machine with the secured kernel. The host machine is equipped with an Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz processor and 10GB of RAM; the running OS is 64-bit Ubuntu 14.04 with Linux kernel version `3.13.0-55`. The virtual machine (VirtualBox) was configured to have a 4-core processor and 4GB RAM; its OS is also 64-bit Ubuntu 14.04. We used the default configuration to compile the kernel code.

Performance with system services. We used LMbench [125] as the micro benchmark to test the runtime slowdown in system services. The selected system services are mainly syscalls, which conform to typical kernel performance evaluations (e.g., [91]). The evaluation results are shown in Table 10. The average performance overhead is only 2%, and in most cases, the performance overhead is less than 5%. These numbers confirm that our zero-initialization-based mitigation for kernel stack is efficient.

Performance with user programs. We further used the SPEC CPU 2006 benchmarks as a macro-benchmark to test the performance impacts of our mitigation over the user-space programs. We ran the test 10 times and adopted the average number. Table 11 shows the evaluation results. Our zero-initialization-based mitigation imposes almost no performance overhead (0.47%) to the SPEC benchmarks on average.

Both the LMbench and SPEC benchmark results confirm that our mitigation is very efficient and reliable (no single error was observed during the evaluation).

Table 11: User space (x86_64) performance evaluation results with the SPEC benchmarks. Time is in second, the smaller the better.

Programs	Baseline	W/ defense	Overhead(%)
perlbench	3.62	3.62	(0.0%)
bzip2	4.74	4.75	(0.2%)
gcc	0.945	0.945	(0.0%)
mcf	2.71	2.68	(-1.1%)
gobmk	13.9	13.9	(0.0%)
hmmer	2.02	2.03	(0.5%)
sjeng	3.28	3.30	(0.6%)
libquantum	0.0365	0.0365	(0.0%)
h264ref	9.35	9.40	(0.5%)
omnetpp	0.342	0.349	(2.0%)
astar	7.77	7.74	(-0.4%)
xalancbmk	0.0611	0.0611	(0.0%)
milc	4.47	4.51	(0.9%)
namd	8.84	8.85	(0.1%)
dealII	10.5	10.6	(1.0%)
soplex	0.0201	0.0201	(0.0%)
povray	0.407	0.417	(2.5%)
lbm	1.66	1.68	(1.2%)
sphinx	1.16	1.17	(0.9%)
Average			**0.47**%

2.2.4 Discussion

Exploitability of uninitialized-use vulnerabilities. Not all uninitialized-use vulnerabilities are exploitable. First, in order to benefit from targeted stack-spraying, the execution path that triggers an uninitialized-use vulnerability must not overwrite the prepared malicious data. Otherwise, the targeted stack-spraying technique will lose control of the uninitialized memory thus cannot exploit this uninitialized-use vulnerability. To verify if the prepared data persists until triggering the uninitialized-use vulnerability, attackers can obtain the address of the instruction using the uninitialized memory and use `kprobes` to intercept the instruction to verify if the prepared data persists. Second, our current deterministic stack spraying does not consider the case in which the preparation of the malicious data occurs in the same syscall that also triggers the uninitialized-use vulnerability. We ensure only that the prepared malicious data persists until the entry point of the syscall triggering the uninitialized-use vulnerability.

Improving mitigation and other defenses. As mentioned in §2.2.3, we can efficiently mitigate uninitialized-use exploits by zero-initializing all pointer-type fields for which the compiler cannot prove that they are properly initialized before reaching a sink (i.e., they are used). This lightweight approach works for most cases. However, false negatives cannot be fully excluded: If a pointer is modified by (or depends on) an uninitialized non-pointer value, zero-initializing this pointer cannot effectively prevent the exploits because the resulting pointer is still controllable by attackers if they can control the non-pointer value. Therefore, one possible improvement for our proposed defense is to zero-initialize non-pointer values as well. Two approaches that already offer a broader defense in this respect are PaX's STACKLEAK [175] and split kernel [97] (see §2.2.5 for details). Both approaches provide strong security to prevent uninitialized-use exploits, but come at the cost of a significant runtime overhead [117]. As such, a sophisticated inter-procedural and field-sensitive analysis is necessary to filter out safe allocations. We leave this challenging problem for future work.

Another defense direction is to defeat targeted stack-spraying. A mitigation against the deterministic stack spraying technique is to randomly adjust the stack base upon syscall entry so that the malicious data prepared in the previous syscall may not overlap the uninitialized variable in the vulnerable syscall. Since the kernel stack has only 8KB or 16KB, the entropy of such a randomization is limited. To detect the exhaustive memory spraying technique, systems can monitor a large amount of process creations or large memory allocations. However, this spraying technique can be stealthy by reducing the amount of process creations and the size of memory allocations, and probing more times.

2.2.5 Related work

2.2.5.1 Memory Spraying

Memory spraying is a popular means to memory-corruption attacks. By far the most popular memory spraying techniques is *heap spraying*, an attack that was first described by SkyLined in 2004 [150]. Heap spraying attacks fill large portions of the victim's heap memory with

malicious code (e.g., NOP sleds), thus increasing the chance of hitting malicious code for hijacking the control flow [60, 63]. Although the heap spraying technique itself has been countered by the introduction of Data Execution Prevention (DEP), the evolution of heap spraying—*JIT spraying*—has become a popular concept for enabling a variety of web-based attacks [170]. JIT spraying exploits the predictability of the JIT compiler to create predictable code fragments that can be used to hijack control-flow [170, 188]. Since these fragments reside in an executable code cache, mitigation techniques like DEP or $W \oplus X$ can be bypassed [170, 188]. Existing defenses against heap/JIT spraying attacks either try to detect the attack by searching for large amounts of NOP sleds and shell code [60, 63, 150] or randomizes the memory layout and register assignments [47, 60, 188]. Recently, memory spraying has also been used to exploit the "Rowhammer" vulnerability in DRAM devices where repeated access to a certain row of memory causes bit flips in adjacent memory rows [22, 156].

In contrast to all these existing spraying techniques, our targeted stack-spraying target the stack instead of the heap. More importantly, our stack spraying technique is deterministic and stealthy (thus is hard to detect), and our exhaustive memory spraying technique is highly reliable.

2.2.5.2 Kernel Exploits and Automated Exploits

Since the kernel is often a part of the trusted computing base of a system, avoiding exploitable kernel vulnerabilities is critical for the security of a system [34]. Nonetheless, despite the efforts of kernel developers to find and eliminate these vulnerabilities, new such vulnerabilities are still frequently detected. As of the thesis writing, a total of 1,888 vulnerabilities have been reported in the Linux kernel alone, and 348 have been reported in 2017 [141]. With Linux kernel vulnerabilities being on the rise, corresponding exploitation techniques have caught the interests of attackers. One recent approach exploits use-after-free vulnerabilities in the Linux kernel by leveraging its memory recycling mechanism [190],

while another one circumvents existing defenses by manipulating the kernel page table [103].

Although many vulnerabilities and their corresponding exploits are still discovered manually, automatic detection and exploit generation is becoming increasingly popular, as is evidenced by the DARPA Cyber Grand Challenge (DARPA CGC) [171]. In this challenge, teams are required to build automated vulnerability scanning engines, which they then use to compete in a Capture The Flag tournament. One of the tools specifically developed for this challenge is Fuzzbomb [128], which combines static analysis with symbolic execution and fuzzing to detect vulnerabilities in programs. The combination of symbolic execution and fuzzing is also used for the Driller tool [173], which has also been tested on 126 of the DARPA CGC binaries. Driller, like our approach, uses symbolic execution to guide its fuzzing engine in case it fails to generate input to satisfy complex checks in the code. This combination is also used together with static and dynamic program analysis to automatically generate exploits for a wide variety of applications [176]. Similar to these approaches, we also use a combination of symbolic execution and fuzzing to discover execution paths that can achieve targeted spraying in the Linux kernel.

2.2.5.3 Uninitialized Use Exploits

Despite the fact that uninitialized-use bugs are seldom considered to be security-critical, a number of exploits for these vulnerabilities have become known in recent years. Flake [67] used a manual approach towards exploiting uninitialized local variables on the user-space stack, while Cook [41] used an unchecked `copy_from_user()` call with an uninitialized variable to exploit the Linux kernel and gain root privileges. Jurczyk in turn exploited CVE-2011-2018, a stack-based uninitialized-variable reference vulnerability in the Windows kernel, which allows an attacker to execute arbitrary code with system privileges [85]. Last but not least, Chen exploited an heap-based uninitialized-use vulnerability in Microsoft's Internet Explorer (CVE-2015-1745) using fuzzing [32]. Unlike these ad-hoc attacks, our targeted stack-spraying is general and automated.

2.2.5.4 *Uninitialized Use Detection and Prevention*

Researchers have proposed some detection mechanisms for uninitialized uses; however, only few defenses against uninitialized uses have been proposed. For detection, tools such as kmemcheck [138], Dr.Memory [25], and Valgrind [162] leverage dynamic instrumentation and analysis to track memory accesses while compiler-based approaches like MemorySanitizer [172] and Usher [191] insert tracking code to find uninitialized uses at runtime. For defense mechanisms, Kurmus and Zippel [97] proposed a technique for preventing exploits of memory-corruption vulnerabilities. Their approach relies on single-split kernels where system calls of untrusted processes can only access a hardened kernel version while trusted processes can access the unmodified kernel. A solution that is specifically targeted towards uninitialized-use vulnerabilities is offered by the PaX team, known for the invention of ASLR. Their GCC compiler plugins, STACKLEAK and STRUCTLEAK, clear the kernel stack on kernel-to-user transitions and initialize all local variables that might be copied to user space, which effectively prevents uninitialized uses of kernel memory [175]. A key difference of our efficient defense against uninitialized uses is that instead of initializing all local variables, we specifically initialize pointer-type fields that have not been properly initialized before. While STACKLEAK and Split kernel introduce a significant performance overhead (e.g., STACKLEAK introduces an average of 40% runtime overhead in system operations [117]), our lightweight defense imposes almost no performance overhead.

2.2.5.5 *Memory Safety Techniques*

Memory-corruption errors such as dangling pointers are a long-known problem in unsafe programming languages like C. In the last decade, researchers have proposed several approaches to mitigate the exploits of these errors. Watchdog [130] and its successor WatchdogLite [129] both leverage hardware supports to store and check allocation metadata to prevent use-after-free vulnerabilities. Softbound [131] and CETS [132] are software-based approaches that aim to prevent memory-corruption errors at compile-time. Softbound

enforces spatial memory safety by storing base and bound information as metadata for every pointer, while CETS enforces temporal memory safety by storing unique identifiers for each object, which are then used to check if the object is still allocated upon pointer dereferencing. Notably, although these memory safety techniques claim full memory safety, they currently do not cover uninitialized use as a prevention target. In contrast to these metadata-based approaches, DieHard [17] and its successor, DieHarder [139] both focus on randomizing the location of heap objects to make dangling-pointer dereferencing hard to exploit. Since both techniques focus on heap objects, they cannot detect and prevent uninitialized-use errors on the stack. StackArmor [37] also adopts randomization to achieve the memory safety for stack. All these randomization-based memory-safety techniques are probabilistic.

CHAPTER III

RETROFITTING SYSTEM DESIGNS AGAINST INFORMATION LEAKS

Software systems also have insecure designs that may cause information leaks. This is because system designers often prioritize performance and flexibility over security. Examples of such insecure designs include the ASLR-violating profess forking mechanism [20], timing side channel caused by deduplication and Copy-on-Write [22], and AnC (CPU cache side channel) [73]. Unlike eliminating information-leak vulnerabilities that can be achieved with code analysis and instrumentation, fixing insecure designs is challenging because it requires deep understanding and redesign of system mechanisms. Our second way to prevent information leaks is redesigning insecure system mechanisms. Since there are no uniform patterns for insecure designs, we have to fix them case by case. In this section, we particularly focus on the ASLR-violating profess forking mechanism in daemon servers (e.g., web servers), which allows attackers to break ASLR efficiently. We propose the runtime re-randomization mechanism to ensure that the forked processes have their own randomized memory layout.

3.1 RUNTIMEASLR*: Re-Randomizing Memory Layout at Runtime*

In the arms race of remote code execution, the introduction of non-executable memory has relocated the battlefield: attackers are now forced to identify and suitably reuse existing code snippets, while protective technologies aim at hiding the exact memory location of these code snippets by means of various forms of memory randomization. As a result, a variety of different memory randomization techniques have been proposed that strive to impede, or ideally to prevent, the precise localization or prediction where specific code

resides [12, 21, 71, 92, 105, 189]. Address Space Layout Randomization (ASLR) [144, 146] currently stands out as the most widely adopted, efficient such kind of technique.

All existing techniques for memory randomization including ASLR are conceptually designed to perform a single, once-and-for-all randomization before or at the process' load-time. The efficacy of such upfront randomization techniques hence crucially relies on the assumption that an attacker has only one chance to guess the randomized address of a process to launch attack, and that this attack succeeds only with a very low probability. The underlying justification for this assumption is typically that an unsuccessful probing of the attacker will cause the process to crash, so that it has to be restarted and hence will be randomized again. Technically, the assumption is that each attacker's guess corresponds to an independent Bernoulli trial.

In reality, however, this assumption is not valid in many scenarios: daemon servers (e.g., Apache or Nginx web servers, and OpenSSH) `fork()` child processes at runtime which then inherit the randomization of their parents, and thereby creating clones with the same memory layout. Consequently, if a child crashes after unsuccessful probing, a clone with the same address space layout is created again. Therefore, an unsuccessful probing does not require an attacker to start from scratch, instead he can reuse the knowledge gained in his previous probings. In the following, we refer to such attacks as *clone-probing attacks*.

A fundamental limitation with existing ASLR implementations is that they are only performed at load-time. That is, any process that is created without going through loading will not be randomized. For example, all forked child processes always share exactly the same address space layout. As a result, clone-probing attacks enable an adversary to bypass current memory randomization techniques by brute-force exploration of possibilities. Moreover, practical scenarios often exhibit side-channel information that can be used to drastically reduce the search space. In these cases, the consequences are far worse, as impressively demonstrated by recent sophisticated attacks against memory randomization. For instance, Blind ROP [20] exploits a buffer overflow vulnerability to overwrite return

pointers only by one byte each time, and thereby reducing the search space to mere 256 possibilities since the remaining bytes of the return pointer are left unmodified. After a successful return, i.e., if the program does not crash or exhibit unexpected behavior, the first byte of a valid pointer has been correctly discovered. The technique is then used repeatedly to discover remaining bytes. On average, the probing of a byte is successful after 128 tries. This approach hence reduces the brute-force complexity from 2^{64} to $256 \cdot 8 = 2,048$ tries (for 64-bit systems). A successful clone-probing attack can hence be leveraged to bypass ASLR, which is a general prerequisite for further, more severe attacks, including code reuse attacks, privilege-escalation by resetting `uid`, and sensitive data leaks.

The root cause of clone-probing attacks is that the forked child processes are not re-randomized. Starting a process from its entry point by calling `execve()` in the child after `fork()` [20], however, alters the intended semantics of child forking: `fork` is intentionally designed to inherit the parent's state (variables, stack, allocated objects etc.), whereas `execve()` starts another, fresh instance of a program so that the execution starts at the program's entry point without inheriting any information from its parent. It is *possible* to make the forked child process independent from the parent process so that `execve()` can be used to easily re-randomize the child process; however, the child process will not be able to benefit from the semantic-preserving and resource-sharing (e.g., sharing file descriptors and network connections) `fork()`, and the target server program has to be carefully restructured and rewritten to make the child process not depend on any resource or data of its parent process. For example, the complicated dependencies (e.g., on program structure, file descriptors, shared memory, and global variables) made us give up rewriting Nginx (about 140KLOC) to use `execve()`. As a result, we aim to propose a practical (e.g., negligible performance overhead) and easy-to-use (e.g., supporting COTS binary directly without modifying source code) re-randomization mechanism to prevent clone-probing attacks in a semantic-preserving manner.

In this work, we propose RUNTIMEASLR—the first semantics-preserving approach that

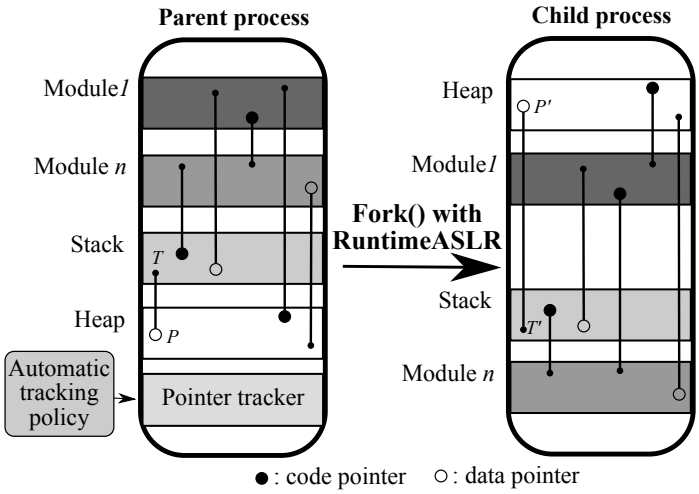

Figure 14: The RUNTIMEASLR approach for re-randomization after fork(). Pointer tracker tracks all pointers for patching after re-randomization. For example, after the re-randomization, since `T` is moved to `T'`, `P` is patched to `P'`.

prevents clone-probing attacks, by consistently re-randomizing the address space of children after `fork()` at runtime while keeping the parent's state. More specifically, the work makes the following contributions: (1) a *semantics-preserving and runtime-based approach* for preventing clone-probing attacks; (2) a systematic and holistic *pointer tracking mechanism* as our approach's central technical building block; and (3) an open-source *implementation* of our approach and a corresponding *evaluation* on Nginx web server.

The RUNTIMEASLR *approach.* Contemporary implementations of the `fork` system call simply clone the parent's address space in order to create a child that inherits the same state. Cloning is cheap in terms of complexity and benefits from inherited open file descriptors and network connections. Instead, RUNTIMEASLR re-randomizes the address space of children after `fork()` at runtime while inheriting their parent's state, as shown in Figure 14. This consistent re-randomization imposes formidable challenges since the state of a program typically incorporates references to a variety of objects, including code, data, stack and allocated objects on the heap. A successful randomization in particular has to ensure smooth continued execution at the new address; hence, it has to relocate code, data and all references consistently at runtime. RUNTIMEASLR meets this challenges by first conducting a pointer tracking policy generation process that has to be run once, followed by a novel pointer

tracking mechanism that is only applied to the parent process. By treating a pointer as a tainted value, the pointer tracking problem is naturally transformed into a taint tracking problem: the initial pointers prepared by OS are used as taint sources, pointer tracking is done using taint propagation, and the necessary propagation policy is automatically created. To generate a pointer tracking policy, instead of manually identifying which CPU instructions typically carry on tainted pointers and which ones remove taints [40, 53, 163], we opted for an automatic approach: the policy generation process executes the target program and sufficient sample programs using Intel's *Pin* instrumentation tool and inspects the semantics of each executed instruction with regard to pointer creation, manipulation or deletion. The result is a policy that describes exactly which instructions deal with pointers in memory and registers.

Pointer tracking mechanism. We use the generated policy to track pointers during runtime. Whenever the parent `fork`s a child, RUNTIMEASLR performs address space rerandomization in the cloned child process and patches pointers according to the tracked pointers of the parent process. As a result, the child processes have mutually different address space layouts without facing any instrumentation overhead. Hence, RUNTIMEASLR imposes no overhead to the services provided by the worker processes of daemon servers. Effectiveness requires us to correctly identify all pointers inside *all* dynamically allocated structures, such as stack, heap, `.bss` or even uncharted `mmap`'ed memory. To this end, RUNTIMEASLR first identifies all pointer sources, which we classify into three categories. First, initial pointers prepared by the OS (e.g., the stack pointer `rsp`) because their location is deterministic and fixed; second, program counters (e.g., `rip`); and third, return values of certain syscalls (e.g., `mmap`, `mremap`, and `brk`). In an ASLR-enabled program, no static pointer is allowed, thus any other pointer must be directly or indirectly derived from these pointer sources. In order to address taint propagation that occurs in the kernel, we identify all syscalls that may propagate pointers and that adapt memory mappings. Our findings show that only a limited number of pointer-propagating syscalls need to be taken into account, and

that the memory-related syscalls (e.g., `mmap`, `munmap`, and `mprotect`) can be hooked in order to track changes in memory maps. With taint policies and appropriate syscall modeling in place, pointer tracking is realized by consistently updating hash tables storing taints.

Implementation and evaluation. We have implemented RUNTIMEASLR based on Intel's Pin on an x86-64 Linux platform. RUNTIMEASLR is implemented as a combination of three tools: a taint policy generator, a pointer tracker, and a randomizer. The taint policy generator and the pointer tracker are implemented as two Pintools; the randomizer is implemented as a shared library that is dynamically loaded by the pointer tracker in the child process. Note that, the current implementation of RUNTIMEASLR is dedicated to server programs that pre-fork worker processes at beginning. For other programs that do not adopt the pre-fork scheme, different implementation approaches (e.g., compiler-based code instrumentation) may be preferred instead of dynamic code instrumentation. More details will be discussed in §3.1.7. To evaluate the effectiveness of RUNTIMEASLR, we applied it to the Nginx web server because it is one of the most popular web servers and has been an attractive attack target in the past [20, 23, 62, 65, 114]. Our evaluation shows that RUNTIMEASLR identifies all pointers, effectively prevents clone-probing attacks. Due to the dynamic instrumentation based, heavy-weight pointer tracking, it takes a longer time for RUNTIMEASLR to start the server programs (e.g., 35 seconds for starting Nginx web server), however, the one-time overhead is amortized over the long run-time, and more importantly, RUNTIMEASLR imposes no performance overhead to the provided services after the worker processes are pre-forked.

3.1.1 Overview of RUNTIMEASLR

RUNTIMEASLR aims to prevent clone-probing attacks by re-randomizing the address space of a process at runtime. More specifically, RUNTIMEASLR protects daemon servers that fork multiple child (worker) processes for responding to users' requests.

3.1.1.1 Threat Model

The attacker's goal is to launch attacks against a daemon server with ASLR enabled, e.g., code re-use attack like return-oriented programming, privilege-escalation by overwriting the `uid`, or stealing sensitive data stored at a certain address. We assume the daemon server consists of a daemon process and multiple worker processes forked by the daemon process. For the sake of robustness, if a worker process is crashed, a new worker process is created by the daemon process with `fork`. This model is widely adopted by daemon servers, e.g., Apache web server, Nginx web server, and OpenSSH. Bypassing ASLR is a general prerequisite of these attacks, as they usually need to access the code or data (at a particular address) in the process memory. To bypass ASLR, the attacker can mount a clone-probing attack to iteratively recover the address space layout, which is shared among all children of the daemon process.

We assume that the operating system running the daemon server realizes the standard protection mechanisms, e.g., $W \oplus X$ and ASLR, and the daemon server is compiled with `-PIE` or `-fPIC`, i.e. the server is compiled to benefit from ASLR. However, the daemon binary may not be diversified, and hence the attacker may have exactly the same copy of it. As a result, attackers can perform both static and dynamic analyses on the daemon program, e.g., scanning it for vulnerabilities. We assume there exists at least one exploitable buffer overflow vulnerability, i.e., a buffer can be overwritten arbitrarily long including return pointers and frame pointers.

RUNTIMEASLR focuses on preventing clone-probing attacks that indirectly infer the address space layout by repeatedly probing worker processes of daemon servers. Other programs that do not adopt the pre-fork scheme usually do not suffer from the clone-probing attacks, and thus are out of scope. Direct leaks, e.g., a leaked pointer in a malformatted `printf` or a memory disclosure vulnerability [159], are out of scope for this work, but have been covered in existing work [11, 48, 50, 119]. Physical attacks, e.g., cold boot attacks, are out of scope. We assume the OS is trusted, so attacks exploiting kernel vulnerabilities are

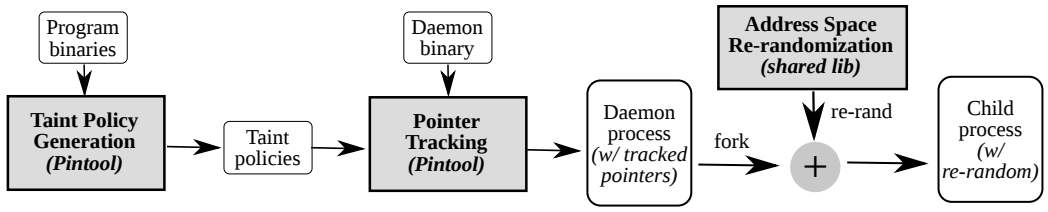

Figure 15: An overview of RUNTIMEASLR's architecture. It consists of three components: taint policy generation, pointer tracking, and address space re-randomization. Program binaries include the binary of the target program.

also out of scope.

3.1.1.2 The RUNTIMEASLR *Approach*

To defeat clone-probing attacks, one can either reload the child process or re-randomize the address space of the child process by patching all pointers. As reloading the process will not inherit any state or semantics from the parent process, it would constitute a great loss as a programming paradigm. Moreover, reloading requires the existing server programs to be thoroughly restructured and rewritten to make the child process not depend on any resource or data of parent process. Therefore, RUNTIMEASLR instead keeps the semantics of its parent process and re-randomizes the address space at runtime.

Since we want RUNTIMEASLR to be a practical tool, it is designed to use user mode *on-the-fly* translation of programs and hence does not require any source code or OS modifications. Since daemon servers usually respond to a multitude of requests at the same time, performance is also a primary goal of RUNTIMEASLR. Those two goals are brought together by dynamic program instrumentation using Intel's *Pin Tool*: it allows to monitor and modify arbitrary programs on-the-fly and can be detached after randomization in order not to impose any performance overhead.

The high-level overview of RUNTIMEASLR is depicted in Figure 15. We first provide an overview of its three main components. Then, technical details will be given in the subsequent sections.

Taint Policy Generation. The first phase of RUNTIMEASLR– taint policy generation –

automatically generates pointer tracking policies by inspecting lots of input programs. The generated policy grows with every input program by "learning" more behaviors of Intel x86-64 instructions. At some point, all possible usages of instructions have been seen and the policy does not grow any further. The generated policy file can then be used in the second phase, pointer tracking, to accurately track pointers of arbitrary programs. Although the identification of pointers is not a new topic, existing approaches lack a systematic approach that can vouch that the used tainting policy is sound. Additionally, prior approaches often aimed at catching out-of-bounds access, which is a slightly different problem as it also needs to know the size of the object pointed to by the pointer [40, 53, 88, 98, 131, 133, 163]. These approaches either rely on type-based analysis, or on heuristic-based taint analysis to identify pointers. Type-based analysis can only identify pointers with pointer-type. However, integers that are also used as pointers, such as base addresses, cannot be identified. Heuristics, on the other hand, do not provide any sound guarantee, since they might incorrectly track identified pointers or fail to identify some pointers in the first place. For example, previous approaches [40, 163] that work on instruction-level tracking only specified a handful of operations (e.g., assignment, addition, subtraction, and bitwise-AND) to carry on taints if the input was tainted. However, as we show in Table 13, there are not only many more instructions that either carry on a taint or remove a taint, but many of them come as a surprise and can only be discovered by a systematic approach rather than empirical heuristics. For example, the `CPUID` and `RDTSC` instructions modify general purpose registers, thereby potentially overwriting stored pointers.

To tackle this problem, we propose an automated taint policy generation mechanism that *automatically* discovers all kinds of instructions that have direct or indirect effect on stored pointers. The basic concept for creating a pointer tracking policy is to inspect all instructions with respect to whether they create, update, or remove a taint. To this end, we check if an instruction writes to a register or to a memory location and whether that written value constitutes a pointer. Then a pointer is determined based on whether its value points

inside any currently mapped memory. This approach does not miss any pointers, i.e., there are no false negatives as long as they point into valid memory. However, this approach might coincidentally identify a random integer value as a valid pointer (false positive). For a 64-bit address space such false positive probability is very low since mapped memory segments are sparse in relation to the huge address space. Nevertheless, we further reduce that probability by running the program multiple times with ASLR enabled, and then compare the results. Correctly identified pointers occur in all runs, whereas false positives do not. We use this technique to automatically create a policy that identifies all instructions that will generate or remove a pointer.

Pointer Tracking. In the second step, the generated policy is used to perform pointer tracking. Pointer tracking itself consists of three parts: (1) it collects all initial pointers prepared by OS; (2) it performs taint tracking for pointers based on the taint policies generated by the first component; (3) since RUNTIMEASLR is explicitly designed to not modify the OS, some syscalls are modeled for taint propagation. The output of this component is the list of all pointers in the process.

Address Space Re-randomization. Using the tracked pointer list, the third component of RUNTIMEASLR– address space re-randomization – then performs module-level memory remapping for all currently mapped memory segments in the child process. As Pin is no longer necessary in the child process, the detachment of Pin is immediately performed after `fork`, and then the re-randomization is triggered as a callback of Pin detachment. After the re-randomization, the dangling pointers are corrected using the list of identified pointers. As a last step, the control is transferred back to the native child process code. Since Pin is already detached, there is no performance overhead in the child processes associated to dynamic instrumentation (see §3.1.6 for performance results).

3.1.2 Automated Taint Policy Generation for Pointers

As alluded to earlier, the existing approaches for taint tracking on binary code are not complete. To tackle this challenging problem, we propose a novel pointer tracking mechanism that accurately identifies pointers (tainting) and tracks pointers throughout their life-cycle (taint propagation). In general, taint analysis can be categorized into *static* taint analysis and *dynamic* taint analysis. In theory, static taint analysis can identify all pointers without false negatives. However, false positives may be unavoidable [59] because of higher level compound type definitions such as `structs` and `unions`, which are not necessarily pointers. As a building block for re-randomization, however, we must neither allow any false positives nor false negatives. Otherwise, an overlooked pointer (false negative) will reference an outdated memory location and will most likely cause a program crash due to an illegal memory access. Similarly, tagging innocent integer values as pointers (false positive) changes data when patching the believed-to-be pointer resulting in erratic program behavior.

Given the limitations of static taint analysis and the strict requirements of our runtime re-randomization, we choose to leverage dynamic taint analysis to identify pointers. Dynamic taint analysis, however, needs a policy that describes which instructions of an entire instruction set architecture (ISA) modify a pointer, either directly or through side effects, and how the pointer is modified. This is possible by studying the 1,513-pages Intel instruction set architecture manual [84] and hoping that one did not overlook a side effect of an obscure instruction. Alternatively, we opted for an automatic learning process that creates annotations for each observed instruction of the ISA by monitoring what each instruction does during runtime. This is possible by leveraging dynamic instrumentation based on Intel's Pin, which makes the execution of each instruction tangible. By carefully inspecting how and which memory and registers an instruction with different operands modifies, a policy is generated that describes how each instruction either creates a taint, removes a taint, does not modify a taint, or even propagates a taint to another register or into memory. This policy grows with every taught program and it eventually can be used in the second phase, the

Pointer Tracking phase, to actually track which parts of memory or registers store pointers. The latter part is described in the next section.

Workflow. The sample programs used for generating the taint policies also contain the target program. We use Pintool to hook each instruction that writes to at least one register or one memory address (Pin provides an API to iterate all operands, including the implicit ones, e.g., operand `EAX` in `RDTSC`). Then, Pintool provides us with the metadata information for that particular instruction. This includes the opcode, count of operands, types of operands (register, memory, or immediate), widths of operands, read/write flags of operands, and data of operands (e.g., register number or immediate value). For conditional instructions (e.g., `cmovnz`), we further include the flag bits in register `rflags`. As an additional step, we hook syscalls to identify the ones that create pointers, e.g. `mmap()`. Also, memory-related syscalls such as `mmap()` are monitored for the requested memory range they allocate because our algorithm needs to know which memory is currently valid and which is not, for pointer verification.

The actual policy generation executes each instruction and compares the state of all registers and accessed memory before execution to after the execution of each instruction. This way, side effects of an instruction can be detected even though the register was not specified as an operand. For example, `RDTSC` reads the CPU's internal Time Stamp Counter (TSC) into `EAX:EDX`, thereby overwriting what was stored in `RAX` and `RDX`. This can be detected by checking each register for being a pointer using multi-run pointer verification (3.1.2.1). If an executed instruction modifies "pointer"-ness of a register or memory location, it is reported as a new policy for that instruction. This policy includes a description whether a pointer was created, removed or copied somewhere else.

Example. The instruction `mov RDI, RSP` (in Intel syntax) is always the first instruction that gets executed in every program. We first extract the opcode (that is `384` in Pin), the types, widths, and read/write flags of its operands. The operands are two 64-bit registers with the first one being destination and the second one being source. Then we analyze whether any

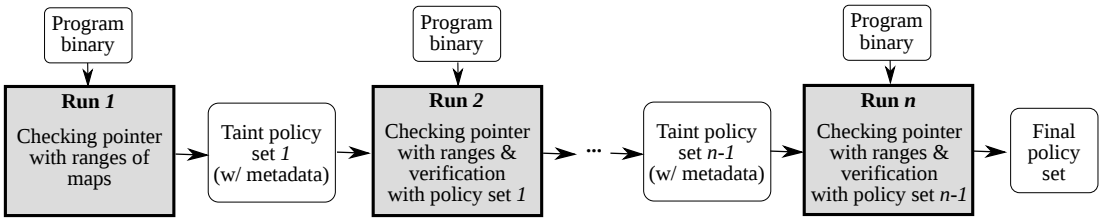

Figure 16: The work-flow of multi-run pointer verification. For each policy, the metadata indicates which registers/memory is written to. Only pointers that point to the same relative address throughout multiple runs are kept.

of those operands (`RDI` or `RSP`) is a pointer using multi-run pointer verification (3.1.2.1). In this case, `RSP` is a pointer, while `RDI` is still zero after initialization of the program. After executing the instruction, the operands are checked again for "pointer"-ness. In this case, `RDI` becomes a pointer after execution. Therefore, we generate a pointer tracking policy as follows. Given an instruction with opcode = `384` and exactly two 64 bit registers as operands, the first operand will be tainted after execution if the second operand was before execution.

3.1.2.1 Realizing multi-run pointer verification

To ensure that only actual pointers are tracked, it is crucial that the pointer detection does not mistakenly classify an integer as a pointer, simply because its numerical value coincidentally represents a valid, mapped memory address. To address this problem, we propose *multi-run pointer verification* to check if a value indeed constitutes a real pointer.

The idea of multi-run pointer verification is inspired by the fact that in a 64-bit address space, the mapped memory is sparsely allocated, and it is unlikely for a random data to point into mapped memory. So by checking if a value points into mapped memory, we can determine a pointer with a high probability. To further decrease the false positive rate, we execute the program multiple times at different load addresses (with ASLR-enabled). The workflow of multi-run pointer verification is shown in Figure 16. In the first run, we output discovered policies with metadata, including the relative location (relative address into the base address of corresponding loaded module) of the targets of generated pointers. For each

of the next runs, only the intersection of policies is kept, which identifies those pointers whose targets have the same relative locations.

The probability that a non-pointer value passes this check and is hence classified as a valid pointer is extremely low. On 64-bit x86 machines, Linux reserves the lower 47 bits for user mode programs[1]. Pointing inside the user mode address space is thus possible if the upper 17 bits of a random 64-bit number are set to zero. Additionally, the lower 47 bits must point into valid mapped memory that has been chosen randomly by ASLR. For the combination of all loaded modules with a total size of b bytes, the probability for a 64-bit integer pointing inside any loaded module is $b \cdot 2^{-64}$. For each run of the multi-run pointer verification, the probability decreases drastically. In the next run, the probability is not only $b \cdot 2^{-64}$ to be inside any loaded module, but the same instruction must produce an integer whose value has the same offset into the same module to be still considered a (false positive) pointer. Hence, only one valid address for the given randomization remains, which has a probability of only 2^{-64} for randomly chosen integers. For `n` runs, the final false positive rate is deceased to $b \cdot 2^{-64 \cdot n}$. The running Nginx web server for example has a total of approximately 22 MB of mapped valid memory. This would result in a false positive rate of $\approx 2^{-103}$ in a `2`-runs verification.

Multi-thread Support. In order to identify the same pointer across different runs, it is necessary to deterministically enumerate all pointer creations in a reliable way – one way would be the order of pointer creation, another would be the memory position of the pointer itself. In a single-threaded program, the position of pointers and the order of their creation are deterministic. However, multi-threaded programs might create pointers in a different order, due to that threads share a common heap, which also influences the pointer positions inside the heap. To overcome this problem, we also consider where pointers are stored. For pointers that are stored on the heap, the base address of the heap object the pointer resides in

[1] `arch/x86/include/asm/processor.h:881` of Kernel 4.1 defines `#define TASK_SIZE_MAX ((1UL « 47) - PAGE_SIZE)`

and its relative position into the base of heap object are recorded and used to uniquely refer to that particular pointer. Since loadable sections (e.g. `.data` or `.bss`) have their objects at fixed locations and each thread has its own stack, they are not affected by multi-threading.

3.1.3 Taint Tracking for Pointers

Using the taint policy that has been created *once* in the taint policy generation step, the program can be run again with pointer tracking enabled based on the created policy. The first step is to *identify* a pointer when it is created. It is crucial to identify exactly *when* a pointer is created to be able to track it from its inception. Otherwise, it might have been copied to other places already without noticing.

3.1.3.1 Pointer Sources

In ASLR-enabled programs, no static pointer (known at compile time) can exist because all code and data are loaded at an unpredictable address. All pointers are either derived from the current address of execution (`RIP` register), the stack (`RSP` register), the heap (call to `mmap()`, `mremap()` or `brk()`) or they are injected by the OS. For example, `lea RAX,[RIP+0x2007bc]` derives a pointer based on current instruction pointer (`RIP`), and saves it in the register `RAX`. In fact, `RIP` and `RSP` are also injected by the Linux kernel before the program starts. In chronological order, whenever a new process is loaded after calling `execve()`, the Linux kernel inserts pointers into the process. As per the Linux 4.1 source, `execve()` first clears all registers[2] and then sets the first instruction to execute by modifying the `RIP` register and setting the stack register `RSP` to the end of the stack. Then, initial environment variables and program arguments are pushed to the stack before the first instruction of the newly created process begins execution. The initial data in stack prepared by OS also contains some pointers (e.g., entry point). As OS routinely stores these initial pointers in stack in the same order at load-time, their relative locations are fixed, we also apply the idea of multi-run pointer verification to accurately identify them. Once all of these pointer sources

[2]Macro `ELF_PLAT_INIT(regs, load_addr)` of file arch/x86/um/asm/elf.h

are found and tainted, the discovered taint policy of the earlier step can then accurately tracks other pointers derived from these pointer sources.

3.1.3.2 Syscall Modeling

The only other way to introduce new pointers is as a result of a syscall. Since we do not want to track pointers in kernel mode, we rather use well-established method *syscall modeling* to mimic the behavior of the kernel with respect to pointer tainting in user mode. Our analysis of all side effects of syscalls has revealed that the only syscalls that modify mapped memory and hence create new pointers are `mmap`, `mremap` and `brk`. Those syscalls are monitored during execution by hooking them inside the Pintool. The return values of those syscalls are then tainted as pointers accordingly. As a side effect, the gained knowledge about memory mappings is used as a plausibility check that data tainted as pointers indeed points into valid, mapped memory.

3.1.3.3 Bookkeeping

The internal bookkeeping of which register or memory location is tainted is stored in a simple but fast hash map. Indexing the memory hash map with an address will return taint information about that address if applicable. Untainted memory will of course not be stored in the hash map and returns `null`. According to the used taint policy, each executed instruction may propagate a taint into another register or memory location (e.g., `mov` or `add`) or remove the taint (e.g., `xor RAX, RAX`). The taint policy contains tuples describing matches based on their opcode, operand type (i.e., register, memory, or immediate), operand size, flag bits (for conditional instructions), and whether operands are already tainted. Using this tuple, the policy is queried on the expected action for the modified registers or memory. If a matching policy is found, it is applied, i.e., the expected registers or memory locations are tainted or untainted. Otherwise, no pointer propagation effect is expected and no taints are changed.

3.1.4 Address Space Re-randomization

Even with the knowledge of exact pointer locations, re-randomization of a running process is not straight-forward. In fact, we faced the following challenges:

C1: The child process also inherits the dynamic instrumentation of the Pin by `fork()`. For performance reasons, Pin in child process is unnecessary, thus should be detached.

C2: Unmapping and clearing of Pintool is a chicken-and-egg problem since the instrumented `fork()` code would need to unmap `itself` while executing.

C3: Remapping a set of memory blocks is not an atomic process and hence during the intermediate state of remapping, no valid stack and no valid library functions are accessible as they are already remapped.

To overcome those challenges, we designed our re-randomization mechanism as a separate shared library as shown in Figure 17, which is triggered by the callback of Pin's detaching. This way, it can (1) be attached to child process, (2) perform the re-randomization and patch all pointers,(3) clear the Pin code/data, and finally (4) load updated registers and perform a context switch to the newly `fork`'ed child.

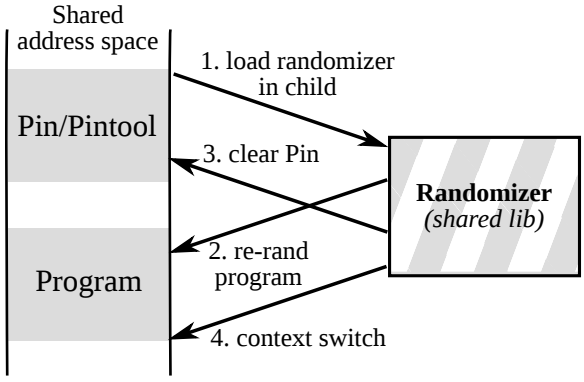

Figure 17: The work-flow of re-randomization.

3.1.4.1 Re-randomization Granularity

The default ASLR provided by Linux OS performs randomization in process-level, which means offsets between modules are fixed. RUNTIMEASLR chooses to perform module-level randomization by specifying random base to `mmap`, which achieves a better effective entropy [78]. However, finer-grained randomizations [12, 56, 71, 80, 92, 187] can be further applied thanks to the tracked pointer information.

3.1.5 Implementation

RUNTIMEASLR is implemented as a combination of three tools: the taint policy generator, pointer tracker, and randomizer. The taint policy generator and pointer tracker are implemented as two Pintools for `pin-2.14-71313`, while the randomizer is implemented as a shared library that is dynamically loaded by the pointer tracker into the child process.

3.1.5.1 Policy Generator

Instruction Abstraction. In the automated policy generation phase, our Pintool learns how x86 instructions "behave" given a certain combination of operands. For this purpose, we are only interested in how they behave with respect to pointer creation, modification, erasure, and propagation. Therefore, the created policy is an abstraction of instruction behavior that is just enough to accomplish accurate taint tracking. Our experiments have shown that, in most cases, it suffices to abstract an instruction to only its opcode, number of operands, type of operands (immediate, register, memory), width of each operand (in bits), flag bits (only for conditional instructions), and which operands were tainted before execution of the instruction. For this exact combination, an entry in the policy describes how it behaves with respect to tainted registers and memory after executing the instruction. For example, let the observed instruction be `mov RDI, RSP` with two operands of 64 bit register type. The second operand `RSP` is tainted, because it is a stack pointer. One specific policy entry for this exact combination (opcode=`mov`, operands=`register64,register64`, taint=false, true)

specifies that after execution the first operand will be tainted. This makes sense, as `mov` actually copies the value of the second operand into the first operand.

Table 12: Ambiguous policies found in tested programs. An implicit instruction may have non-deterministic taintness outputs. 1: pointer; 0: non-pointer.

#	Instruction	Input	Output	Category
1	`add reg1, reg2`	reg1=1 and reg2=0	reg1?=1	arithmetic
2	`sub reg1, mem1`	reg1=1 and mem1=0	reg1?=1	arithmetic
3	`add mem1, reg1`	mem1=0 and reg1=0	mem1?=1	arithmetic
4	`add reg1, mem1`	reg1=0 and reg2=0	reg1?=1	arithmetic
5	`and reg1, imm1`	reg1=1 and reg2=0	reg1?=1	bitwise

Ambiguous Policy. There are special cases for which the aforementioned tuples of operand types, amount of operands, operand sizes and taints are not unambiguous enough to make a statement about taints after the execution of an instruction. Table 12 lists the ambiguous cases found in the tested programs. Case 5 is easy to understand, since a bitwise `and` operation could be used to compute either the base address (i.e., a pointer) or an offset (i.e., a non-pointer) depending on the value of the other operand. For example, assume `RAX` in `and RAX, 0x????????` contains a pointer and is therefore tainted. If the bitmask of the second operand exhibits many set MSBs, e.g., `and RAX, 0xfffffffffffff000`, the result will be the base address of the provided pointer in `RAX`. Vice versa, setting a few LSBs (e.g., `0x0000000000000fff`) calculates the offset into something. Our experiments with real-world programs show that the only use of a bitwise `and` instruction with a pointer is to either calculate the base or an offset. This confirms similar findings by [40] and [163].

Case 1 - case 4 indicate that ambiguous policies also exist for arithmetic instructions – a pointer could be added or subtracted by an offset to generate either a new pointer (e.g., relocation) or a non-pointer. To understand these special cases, we performed an in-depth analysis (on Ubuntu `14.04.2`). Note that RUNTIMEASLR provides a debugging mode that can print context information of executed instructions to ease the analysis.

Case 1. At the beginning of execution in loader, instruction `add r12, rcx` adds the pointer saved in `rcx` with a constant `0x3800003d8`, which generates a "non-pointer" (it does

not point to mapped memory). However, we found that this non-pointer is added back to generate a new pointer with constant -6FFFFEF5×8 in instruction `mov qword ptr [r12+rax×8], rdx`.

Case 2 - 4. We found that there is an instruction `sub rdx, qword ptr [r12]` that destroys a pointer by subtracting it with an offset. Before execution, `rdx` points to the base of `vDSO`, which is then subtracted by constant `0xfffffffffff700000` to generate a value pointing to unmapped memory. Interestingly, we found case 3 and case 4 are paired to case 2, which re-assemble the pointers from the destroyed one in case 4. For example, case 3 is the instruction `add qword ptr [r12+0x348], rdx` that adds the destroyed pointer by constant `0xfffffffffff700fbd` to generate a new pointer still pointing to `vDSO`. We believe the arithmetic operations in case 2, 3, and 4 are used to perform a very simple pointer protection employed by `vDSO`, which is however not found in other normal programs.

Based on the analysis, we generally cope with these ambiguous instructions by the simple range-checking approach: we check whether the result after executing the instruction points into valid memory, if it does, it is a pointer; otherwise, it is a non-pointer. The check only needs to be lazily performed for those ambiguous cases that are clearly marked in the policy file. For ambiguous cases with bitwise operations, range-checking can easily differentiate between a base address and an offset. For ambiguous ones with arithmetic operations, no matter what unrecognizable representation a pointer is transformed into, it will be recognized again by range-checking if it is changed back to the pointer. Therefore, the simple range-checking approach can generally handle all these ambiguous cases in a lazy manner.

Hidden Pointers in SSE Registers. Streaming SIMD Extensions (SSE) is an x86 instruction set extension that provides 128 bit registers `xmm0-xmm15` and and 256 bit registers `ymm0-ymm15`. In theory and good practice, 64 bit pointers of the 64 bit Intel x86 architecture should be stored in its general purpose registers (`RAX, RBX, ..., r1, r2, ...r15`) or in segment registers (`FS, GS`). However, our policy generation showed that these 128 bit SSE

registers may also be used to store pointers, e.g., Nginx web server. Fortunately, those pointers were always stored either in the first 64 of the 128 bits (upper half) or in the last 64 bit (lower half). Based on this observation, RUNTIMEASLR treats a SSE register as a multiple of non-overlapping 64 bit general purpose integer registers. Whenever these SSE register are used, each 64-bit of their content is checked by RUNTIMEASLR for 64 bit pointers.

Mangled Pointers. Interestingly, the GNU standard C library (glibc) that is linked to every C or C++ program performs pointer obfuscation for some syscalls. It is implemented as a simple XOR operation against a random but fixed value stored relative to the segment register `FS` (see Figure 18).

```
#define PTR_MANGLE(reg) \
        xor %fs:POINTER_GUARD, reg; \
        rol $2*LP_SIZE+1, reg

#define PTR_DEMANGLE(reg) \
        ror $2*LP_SIZE+1, reg; \
        xor %fs:POINTER_GUARD, reg
```

Figure 18: Pointer mangling and demangling in libc. The XORing key is hidden by using `fs` segment register. LP_SIZE is 8 on 64 bit platform.

To cope with mangled pointers, we demangle the pointers and check if the demangled values point inside valid mapped memory. This is possible because the XOR value is accessible by the process itself (stored at `fs:POINTER_GUIARD`), but not accessible for attackers residing outside the process. Should the demangled value reveal a pointer, the stored taint is augmented by meta information indicating a mangling according to glibc's method. This meta information leaves room for future, other methods of pointer obfuscation to be implemented. Currently, all tested programs compiled with gcc only exhibit the pointer obfuscation using the `PTR_MANGLE` macro shown in Figure 18.

3.1.5.2 Pointer Tracker

The pointer tracker itself is also implemented as a Pintool. As alluded to earlier, the only initial pointers after process creation are `RSP`, `RIP`, and some initial pointers in stack. In particular, we found `71` initial pointers in stack with the help of multi-run pointer verification. As the kernel routinely loads programs in the same way, we confirmed that these initial pointers have fixed relative locations in stack. From these initial pointers, further pointers are derived using the generated policy, which is applied to the affected instructions specified in the policy set. The policy set is loaded into an STL `list` container. We then use Pin's functionality to hook instructions based on their opcode, operand types and widths of operands. The hooking is realized in Pin by inserting a call before and after each instruction in question. Control flow is then diverted from the main executable code or library code into Pin and thereby into our pointer tracking mechanism. The pointer tracking mechanism fetches information about the used operands (register and/or memory), especially whether they are tainted and their current value. After applying the policy, which results in further or fewer taints for registers or memory, execution continues normally at the original position in memory. Unmatched instructions will not be hooked and execute normally.

Syscall Handling. Since RUNTIMEASLR is designed as a user space tool, all syscalls are a black box operation to RUNTIMEASLR. However, we are only interested in whether a syscall modifies a pointer or creates a new one. We found that most syscalls actually do not generate new pointers or propagate pointers. We manually evaluated the visible effects of the 313 Linux syscalls in user mode. Only `15` of them generate or propagate user mode pointers. Some representative ones are detailed as below.

- `mmap` creates new memory space and returns the base. We taint its return value when it returns.
- `munmap` unmaps the given memory space. In this case, we untaint the corresponding pointers that point to the unmapped memory.

- `mremap` moves an existing memory space to a new one and returns the new base address. In this case, we update the corresponding pointers pointing to the old memory space with the moved offset. Also we taint the returned value.

- `mprotect` changes the access permissions of memory. We update the tracked pointers according to how permissions are changed. For example, if the memory is changed from readable to non-readable, we untaint the corresponding pointers.

- `brk` either obtains current program break (by providing the first parameter with 0) or changes the location of program break. In the former, we simply taint its return value; while in the latter, if it shrinks the memory, we untaint the corresponding pointers and also taint its return value.

- `arch_prctl, etc`. There are also some syscalls that may load existing pointers of user space. For example, `arch_prctl` can be used to get base values in segment register `fs` or `gs`. For these cases, we taint the obtained pointers in return value or parameters.

Pointer Scope and Memory Deallocation. Whenever a called function returns, the local variables of the callee are no longer valid, i.e., they have left their scope. Such local variables may also contain pointers. Hence, it is vital to remove any associated taint of pointers that have gone out of scope. The memory region that contains variables that have gone out of scope is considered uninitialized memory, since reading from that memory is non-deterministic as it depends on which function was called last. A good compiler will catch that type of uninitialized access at compile-time and warn the developer about it.

To prevent unpredicted behaviors, we detect out-of-scope pointers when a function returns so that they can be untainted. To this end, we hook the `ret` instruction and on each return calculate the currently valid stack frame by inspecting `RSP`. By the x86-64 calling convention, the stack pointer `RSP` must point to the top of the valid stack after returning from a function, therefore, all data stored above `RSP` can be untainted.

The same applies to objects on the heap: after a call to `free()` (`delete()` also calls `free()`), pointers stored in that now deallocated area become invalid. We therefore remove all taints of pointers that are stored in a `free`'ed area. A caveat is that the call to `free` does not include any information about the size of the to-be-deleted object. However, the memory in question was once allocated using `malloc` with a specific size. We therefore use a map that associates a length obtained from hooking `malloc` to each heap pointer. Whenever, `free()` is called, we look up the size for the specified base pointer and untaint all pointers stored in the deallocated area.

3.1.5.3 Randomizer

The randomization is done in module-level, which is better than the "process-level" randomization provided by Linux' ASLR implementation. Finer-grained randomization is possible, as we have all pointer information. However, even position-independent code assumes fixed offsets between code and data. Finer-grained randomization may require patching these offsets, and thus introduces more overhead. The actual remapping from an old address (cloned from the parent) to a new random address is achieved by calling the syscall `mremap` with a random base address. We obtain the cryptographically secure randomness from `/dev/random`. In the current implementation, we set the default entropy to the one provided by default Linux OS, which is 28 bits. However, we also provide a configuration switch for the entropy so it can be set up to 48 bits (the full canonical address space on Intel x86-64 platforms). Note that when `heap` is remapped, we also need to adjust `program break` by calling the syscall `sbrk()`. The protection flags and size of the new mapping are simply taken from the old one to preserve all semantics. After remapping, we patch the pointers accordingly (in both memory and registers).

The final step is context switching from Pin to program. The context includes not only general registers but also SSE registers (e.g., `xmm0-xmm15`) and segment registers (e.g., `fs and gs`). As all pointers are patched when the address space is re-randomized, the context

is naturally updated accordingly. Before Pin's mappings are unmapped, the context data is copied to the stack of the program. To load the context into the corresponding registers (after unmapping Pin), we have to use handwritten assembly, as no library functions are available and we cannot even use Pin's stack and heap. Special care must be taken for loading values from stack into SSE registers (i.e., `xmm0-xmm15`) as those memory addresses must be 16-byte-aligned in order to not trigger a general protection fault (`#GP`). Finally, we use an indirect `jmp` to transfer the control back to program and continue the original execution of child process. At this point, Pin is completely unmapped and as the child process is no longer instrumented, it runs with its native performance.

Recursive Forking. So far, we have described that `fork`'ed children are no longer instrumented in order not to suffer from performance penalties. However, one cannot know if child processes will or will not fork their own children. If they do, there would not be any pointer tracking information because it was intentionally disabled for performance reasons. To overcome this, we provided a feature that records which layer of processes of a program typically `fork` their own new processes. For example, the Nginx web server first forks a daemon process, and then that daemon process forks worker processes for each CPU core. Therefore, we can configure which layer of children will disable their instrumentation which will not.

3.1.6 Evaluation

In this section we evaluate the **correctness**, **effectiveness**, and **efficiency** of RUNTIMEASLR. To this end, we have developed the following four evaluations:

- A *theoretical* and *practical* analysis of the accuracy of identifying pointers.
- A memory *snapshot* analysis to compare the correctness of re-randomization with the help of load-time randomization.
- An empirical test of real clone-probing attack prevention.

- A performance evaluation of re-randomized child processes and microbenchmarks of `fork` overhead and pointer tracking in the parent process.

Experiment setup. We applied RUNTIMEASLR to the Nginx web server, as it is one of the most popular web servers and has been an attractive attack target [20, 23, 62, 65, 114]. We chose Nginx in version `1.4.0`, as this version contains a stack buffer overflow vulnerability that was also exploited by Hacking Blind [20]. This way, we can later show the effectiveness of our solution in the presence of a real vulnerability. We use Nginx web server to evaluate the correctness, effectiveness, and performance of RUNTIMEASLR. Further, we applied RUNTIMEASLR to SPEC CPU2006 benchmarks to evaluate the performance of the pointer tracking component. All experiments were conducted on a Dell OptiPlex 390 equipped with an Intel® Core™ i3 2120 running at 3.30 GHz and 8 GiB of RAM. This particular processor has two physical cores each of which supports *Simultaneous Multithreading*, which presents itself as four cores to the operating system. For operating system, we used an unmodified fresh install of the latest Ubuntu long-term support release `14.04.2`.

3.1.6.1 Correctness

We first evaluated the correctness of RUNTIMEASLR by analyzing its accuracy in pointer identification.

Theoretical Analysis. In this section, we present an upper bound for false positive pointer detection. As already described in §3.1.2.1, the probability for a random integer to be mistaken as a pointer is $p = b \cdot 2^{-64 \cdot n}$ for b bytes being mapped into a process' address space and `n`-runs are performed in multi-run pointer verification. Of course, this only applies to a single pointer identification. As every executed instruction is inspected for potential pointers, in the worst case, every inspected instruction might introduce a new random integer that could be mistaken for a pointer. So in the worst case, the entire mapped memory only consists of instructions that handle 64 bit integers. Even though it is technically not possible due to encoding, a maximum of b instructions can exist in b bytes of memory. Therefore,

the probability of identifying at least one integer coincidentally as a pointer is

$$1 - (1-p)^b = b \cdot p - \binom{b}{2} \cdot p^2 + \binom{b}{3} \cdot p^3 - \cdots - p^b$$

according to the binomial probability. Assuming $b < 2^{47}$ and `n` $>= 2$, then $\binom{b}{i} \cdot p^i$ must be significantly larger than $\binom{b}{i+1} \cdot p^{i+1}$, where $1 < i < b - 1$ and $p^b > 0$. Therefore, this probability can be safely simplified to:

$$b \cdot p = b^2 \cdot 2^{-64 \cdot n}$$

To fill in some numbers, we can assume that 100 MB of the address space are used, and `2` runs are performed in multi-run pointer verification. This yields a false positive rate of 2^{-76}, which is negligible.

Memory analysis. Although the protected programs do work successfully, we want to thoroughly check the correctness of RUNTIMEASLR in practice. To this end, we evaluate the false positive and false negative rate of pointers and check if the address space is re-randomized correctly.

False pointer checking. The goal of false pointer checking is to scan the entire address space for 8-byte pointers and make sure that our tracking did find all (no false negatives) and not too many (no false positives) pointers. Checking is done *right before* re-randomization, when it is crucial that all pointers have been tracked correctly. To be fair, multi-run pointer verification was used during the analysis to check false positive pointers. For false negative pointers, we check each 8-byte value in the readable memory to see if it points to mapped memory. We ran the Nginx web server and hooked `fork()` for the memory analysis. This revealed a total of 7952 pointers were tracked with no false positives, i.e. all tracked pointers indeed point into valid mapped memory. The analysis further revealed four "false negatives", i.e. some readable 8-byte values that point into valid memory had not been tracked. However, we found that those *"pointers"* were located in freed heap objects and hence were no longer valid. Since Nginx did not clear the data when freeing heap objects, the out-of-scope pointers remained in memory.

Memory space consistence checking. It is of paramount importance that our forcefully imposed re-randomization at run-time is indeed correct. To verify this, we compare the address space *right after* re-randomization to legitimate address space before forking, but at the same address. This verification process is two-fold. First, the program is run with ASLR enabled so that it is executed at a random address. At the moment a `fork()` syscall is issued, we take a memory snapshot of the entire address space, record base addresses of mappings, and abort the program without re-randomization. Second, we run the program again, which results in a different address, but this time let the re-randomization happen after `fork()` is called. However, the re-randomization of our Pintool is instructed to use the recorded addresses of each mapped block of memory instead of true randomness. This way, the address space of our re-randomization should look exactly like the address space of a properly randomized ASLR process. After that, we compare the remapped memory with the previously dumped memory. Our results show that all mappings are exactly the same, which indicates that all pointers are correctly patched. Please note that in the first ASLR run, the program is run with Pin enabled because our comparison (re-randomization) will have Pin enabled as well. Otherwise comparisons would differ in the mapped memory of Pin.

Some interesting policies. Unlike previous pointer tracking approaches (e.g., [40, 163]) that only empirically cover a handful of taint policies (e.g., assignments, addition/subtraction, and bitwise `AND`), we have automatically discovered a total of **342** taint policies that operate pointers. Table 13 lists some that we think are interesting for the reader.

3.1.6.2 Security

Address space analysis. RUNTIMEASLR performs module-level re-randomization on the child process. RUNTIMEASLR provides `mremap` with cryptographically secure randomness obtained from `/dev/random`. In this evaluation, we set the entropy of randomness to be 28 bits – which is the default entropy of Linux' default ASLR. The generated random base address is of the form `0x7f???????000`, where `?` bits are randomized. The security

Table 13: Selected interesting taint policies that are hard to identify based on heuristics. 1: pointer; 0:non-pointer; mem: accessed memory; ?: ambiguous policy.

#	Instruction	Input	Output
1	`rdtsc`	N/A	rax=0 and rdx=0
2	`cpuid`	rax,rbx,rcx,rdx=1	rax,rbx,rcx,rdx=0
3	`and rcx, rax`	rcx=1 and rax=0	rcx?=1
4	`neg rcx`	rcx=1	rcx=0
5	`add rcx, [rax+0x28]`	rcx=0 and mem=0	rcx?=1
6	`mov eax, edi`	rax=1 and rdi=1	rax=0
7	`rol rax, 0x11`	rax=1	rax=0
8	`lea rdx, [rip+0x21e3cc]`	rdx=0	rdx=1
9	`shr rax, 0x2`	rax=1	rax=0
10	`leave`	rsp=1 and rbp=1	rsp=1 and rbp=1
11	`movdqu xmm8, [rsi]`	xmm8=10 and mem=00	xmm8=00
12	`pslldq xmm2, 0x5`	xmm2=01 and mem=00	xmm2=00

provided by RUNTIMEASLR is the re-randomization of child processes' address space, which consists of two parts: (1) remapping all modules of protected program to randomized addresses; (2) all other modules (e.g., of Pin and Pintool) are unmapped. To evaluate the security, we run Nginx web server with RUNTIMEASLR. We configured the number of worker processes to be 4, which is the default number suggested by Nginx and is exactly the amount of physical CPU cores of the test machine. We then verify the re-randomized memory mappings of each worker process by checking `/proc/worker-pid/maps`. We empirically confirmed that the question-marked bits in `0x7f???????000` are indeed continuously randomized. The relative addresses between different modules are also randomized, which further improves the effective entropy [78] of default Linux ASLR. We also confirmed that all other mappings (of Pin and Pintool) are not shown in `/proc/worker-pid/maps`, and hence have been completely unmapped. Based on these results, we conclude that the expected security is indeed achieved by RUNTIMEASLR.

3.1.6.3 Real Exploit Testing

To further verify the effectiveness of RUNTIMEASLR, we also test if RUNTIMEASLR can defeat real clone-probing attack. Fortunately, the Hacking Blind attack (`BROP`) [20] is

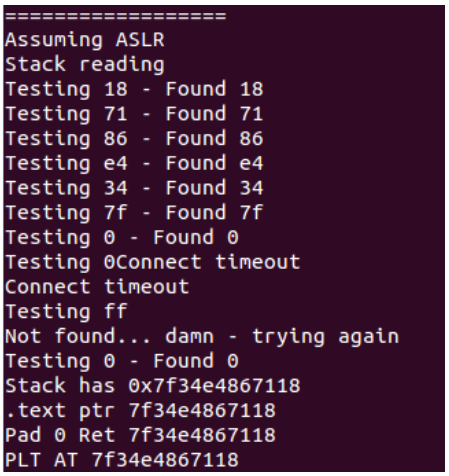

Figure 19: Nginx without RUNTIMEASLR. The clone-probing attack, with the stack reading technique of Hacking Blind, succeeds.

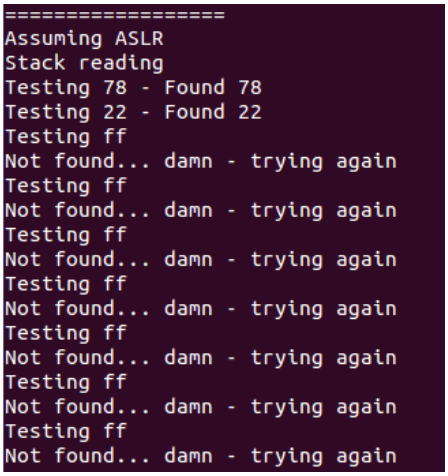

Figure 20: Nginx with RUNTIMEASLR. The clone-probing attack is failed because memory layout is re-randomized after crashing.

publicly available[3]. We downloaded the provided exploit and ran it without any change against `Nginx 1.4.0`. As per `BROP` website, the exploit consists of several steps, including ASLR bypassing with stack reading, locating the required gadgets, and building the ROP shellcode. The foremost step is to efficiently bypass ASLR. The approach proposed in `BROP` is so-called stack reading. As already mentioned in §3.1, it overwrites the return address on stack byte-by-byte with a single guessed byte. As the newly forked worker process has the same address space as the crashed one, the probing can be repeatedly performed for each byte.

To verify the effectiveness of RUNTIMEASLR, we ran Nginx with and without RUNTIMEASLR. The results are shown in Figure 19 and Figure 20. In the case of Nginx without RUNTIMEASLR, stack reading can successfully bypass ASLR within two minutes. However, if RUNTIMEASLR is enabled, we had to manually terminate the exploit after 10 hours, as it could not even guess 3 bytes. The reason why stack reading does not work under RUNTIMEASLR is apparent: the correctly guessed bytes keep changing in later trials due to re-randomization.

[3] `http://www.scs.stanford.edu/brop/`

3.1.6.4 Performance

In order to test the performance of our approach, we conducted two independent experiments to be able to measure the performances of the instrumented parent process (with heavy-weight pointer tracking) and uninstrumented child (i.e., worker) processes.

Performance with web benchmarks. To measure the child process performance, we used the `Nginx 1.4.0` web server and measured different performance aspects. The performance slowdown that might potentially be introduced by our RUNTIMEASLR is measured in comparison to an unmodified system as baseline. Nginx used four worker child processes – one pinned to each core. The client to measure network performance was running on a second machine, which was connected to the Nginx web server over a `1000BASE-T` (Gigabit Ethernet) switch using a $10m$ long `CAT 5e` cable. We measured two dimensions, *connection concurrency* performance and *throughput* performance to check if either of them is affected by RUNTIMEASLR.

Connection concurrency measures the round-trip time in relation to the number n of simultaneously connected clients, i.e., the time it takes from the beginning of a request until the complete response has been received from the server. The more clients that are requesting resources from the server at the same time, the longer it takes the server to send back the entire response. To measure the connection concurrency, we used the *Apache Benchmark* (`ab 2.3`) tool. `ab` lets each of the n simultaneous connections request 10,000 documents from the server as fast as possible.

Throughput in turn measures the rate of successfully received bytes per time unit. For this purpose, we let Nginx deliver a 1 GiB[4] file 100 times and averaged the download speed.

As a test bed, we used HTML documents obtained from real web sites. We mirrored the top 10 Alexa [6] websites into our Nginx root document root directory using `wget`. Performance results showed that the response times only loosely correlate to the size of the

[4]We use the notation of a *Gibibyte* (1 GiB $= 1024 \times 1024 \times 1024$ bytes) instead of a *Gigabyte* (1 GB $= 1,000^3$ bytes)

delivered content (see Figure 21). Since our performance slowdown evaluation is interested in the RUNTIMEASLR performance in relation to the baseline, the document size and content did not affect the results as the quotient of $\frac{\text{RUNTIMEASLR}}{\text{baseline}}$ was unaffected. For the remainder of the concurrency tests, we therefore picked a random top 10 web site to test with: Number 6, Amazon.com (442.1 kB of HTML, CSS and JavaScript).

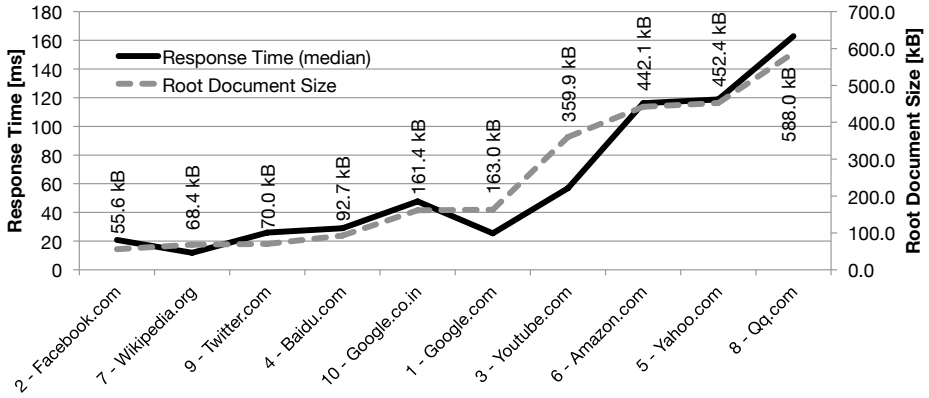

Figure 21: Response time for single real-world documents retrieved from our Nginx web server using 30 concurrent connections. Content mirrored from Alexa Top 10.

Concurrency Results: We tested the concurrency performance of the RUNTIMEASLR-enabled Nginx with 10, 20, 30, 40 and 50 concurrent clients and there was no measurable performance overhead. Figure 22 shows the median of 10,000 response times with the number of concurrent connections $n \in 10, 20, 30, 40, 50$. The worst measured performance difference was for 10 simultaneous connections and is a slowdown of 0.51%, which is well within the jitter margin of the experiment.

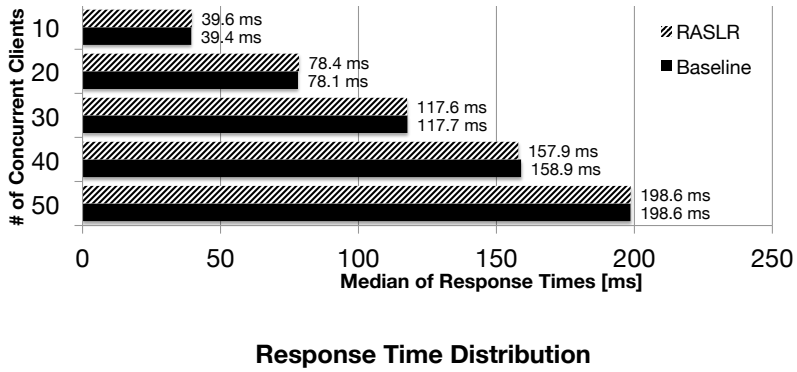

Figure 22: Nginx response time with different concurrencies.

The overall distribution of response times is depicted in Figure 24, which shows that the majority of response times is ≈ 200 ms for 50 simultaneous connections and ≈ 40 ms for 10 simultaneous connections. The performance difference between RUNTIMEASLR and the baseline is not measurable.

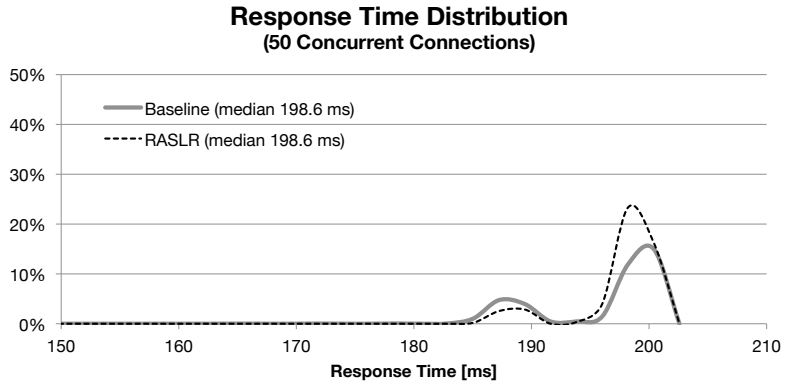

Figure 23: Nginx response time distribution with/without our pointer tracking `Pin` instrumentation. Always 50 concurrent connections

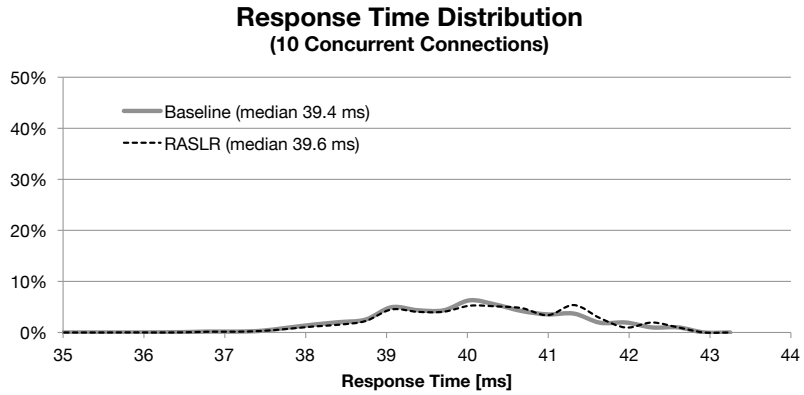

Figure 24: Nginx response time distribution with/without our pointer tracking `Pin` instrumentation. Always 10 concurrent connections

Throughput Results: We also measured the average and peak throughput for a large single file download over HTTP using only one connection at the same time. The throughput result for 100 downloads of a 1 GiB file is shown in Table 14. The measured difference is only 0.5% slowdown on average and a mere 0.1% slowdown for the peak throughput.

Micro benchmark of fork. In traditional UNIX systems and Linux, the `fork()` syscall is tuned to be very fast. We measured the performance overhead of our instrumented `fork`

Table 14: Throughput performance of HTTP file download with RUNTIMEASLR enabled and without (baseline)

	Throughput	
	Average	**Peak**
Baseline	870.5 MBit/s	898.1 MBit/s
RUNTIMEASLR	866.4 MBit/s	897.2 MBit/s

to give the reader an insight of introduced delays. Table 15 shows the micro-benchmarks for the instrumented `fork()` with RUNTIMEASLR. Without RUNTIMEASLR, forking is extremely efficient, taking only 0.1 ms. RUNTIMEASLR performs Pin detaching, address space remapping, Pin unmapping, and context switching for `fork()`. While unmapping and context switching is efficient, the detaching of Pin seems to be a bottleneck. Although percentage of performance overhead is significant, the absolute overhead is less than 150 ms. More importantly, most daemon processes, e.g., Nginx web server and Apache web server, only fork worker processes once and delegate work instead of creating a new process for each work item, so the performance overhead of `fork()` with RUNTIMEASLR only affects the starting time of these servers.

Table 15: Micro benchmark for fork() with RUNTIMEASLR. The original fork() takes 0.1 ms. Worker processes are typically pre-forked once at startup.

	Detach	**Remap**	**Unmap & Context switch**	**Total fork**
Fork w/ RUNTIMEASLR	109.7 ms	27.0 ms	0.8 ms	137.5 ms

3.1.6.5 Performance of Pointer Tracking

RUNTIMEASLR is mainly used to prevent clone-probing attacks targeting daemon servers. As shown in 3.1.6.5, RUNTIMEASLR imposes no overhead to the web service itself. Here, we also want to evaluate the performance of the pointer tracking component. We first measure the time for Nginx web server to start. On our machine, it finishes starting within `35` seconds. We then apply RUNTIMEASLR to SPEC CPU2006 benchmarks which contains some relatively complicated programs, e.g., gcc. Since these benchmark programs do not adopt daemon-worker scheme, pointer tracking will be performed for all executed code. In

Table 16: Pointer tracking on SPEC CPU2006 benchmarks. Pointer tracking is completely detached in child processes, ensuring their efficiency.

Benchmark	**Original** (seconds)	**RUNTIMEASLR** (seconds)	**Overhead** (times)
gcc	1.13	12,783	11,312
mcf	2.64	24,708	9,359
hmmer	2.28	19,004	8,335
libquantum	0.06	1,491	24,850
xalancbmk	0.08	1,932	24,150
soplex	0.02	217	10,850
lbm	2.28	2,468	1,028
sphinx3	1.61	12,661	7,863

this evaluation, all the benchmark programs are compiled with options `-pie -fPIC -O2`. Table 16 shows the results. The significantly reduced performance is not surprising, as RUNTIMEASLR performs runtime taint analysis to accurately identify all pointers. However, pointer tracking is only performed in the parent process at startup. The child process, the actual worker process, is not affected by pointer tracking (see 3.1.6.4). For the long-running daemon servers, the starting overhead introduced by "one-time" pointer tracking before `fork` is acceptable, but the performance of its provided service is more critical, and in our case not affected. One may also wonder about the performance of taint policy generation, although it is less concerned than pointer tracking, since it is performed offline. The running time for taint policy generation against Nginx is about 30 seconds.

3.1.7 Discussion

In this section, we discuss some potential problems with RUNTIMEASLR, which might appear in special cases.

3.1.7.1 Soundness of Taint Policy Generation

The taint policy generation mechanism of RUNTIMEASLR is not sound in general, since the instruction abstraction may generate ambiguous policies. In Section 3.1.5.1, we performed an in-depth analysis to understand how ambiguous policies are introduced in the tested

programs. From that analysis, we learned that normal code emitted by compilers usually process pointers in an universal manner; however, special programs (e.g., dynamic loader and glibc) may process pointers specially, resulting in ambiguous policies (about 1% out of all policies). Although the simple range-checking approach (see Section 3.1.5.1) is sufficient to handle the ambiguous cases listed in Table 12, manual analysis may still be required to confirm the correctness of taint policies when more ambiguous cases are reported by RUNTIMEASLR. If the ambiguous policies cannot be handled by range-checking, one may need to manually define the special taint policies. Alternatively, an automated approach to handle ambiguous policies is to relax the instruction abstraction. For example, including the effective width (or the highest order of bit) of the operands can remove ambiguous case 5 in Table 12. A trade-off needs to be made between the degree of abstraction and pointer tracking performance – relaxing the abstraction will introduce further performance overhead in pointer tracking.

3.1.7.2 Completeness of Taint Policy

The pointer tracking of RUNTIMEASLR consists of the offline taint policy generation and runtime taint tracking. In our experiments, because the sample programs used for generating taint policy include the target program, and the inputs for two runs for policy generation and taint tracking are exactly the same, the policy set is usually complete for pointer tracking in practice – we did not meet any missed policy in Nginx. However, we cannot guarantee the 100% completeness of policy set, as the two runs of program for policy generation and taint tracking are independent. For example, if the system administrator changes the Nginx configurations at runtime (i.e., after policy generation), new code paths not covered during policy generation may be introduced, thus may result in false negative policy. Assuming such cases exist, we can either manually add the reported new policies – as the total number of x86_64 instructions is limited[5] – or adopt an automatic approach: if we find an instruction

[5] http://ref.x86asm.net/coder64.html

generates, updates, or removes a pointer, but is not covered in existing policy set, we can append it to the policy set at runtime.

3.1.7.3 Applicability for General Programs

A program that does not adopt the daemon-worker scheme is not vulnerable to clone-probing, so it is not necessary to use RUNTIMEASLR. As specified in threat model in §3.1.1.1, RUNTIMEASLR is dedicated to server programs that *pre-fork* worker processes processing users' actual requests and perform light-weight tasks (e.g., worker process management) in daemon process, e.g., web servers. RUNTIMEASLR performs runtime taint analysis to accurately identify all pointers in parent (daemon) processes, so the performance of parent process is dramatically reduced. Current implementation of RUNTIMEASLR is not suitable for server programs that perform heavy-weight tasks in parent process for performance reasons. Regarding performance in pointer tracking, RUNTIMEASLR can be improved in two ways: (1) static code instrumentation using a compiler or binary rewriting can help improve the pointer tracking performance. It is worth noting that statically instrumented code is hard to be completely detached in worker processes, which is actually the main reason we chose to employ dynamic instrumentation; (2) as our primary goal is to make the pointer tracking accurate, our current implementation still has room for improvements to tweak its performance. Improving the performance of pointer tracking is a long-studied topic [23, 87, 90], we can employ these techniques to improve the performance of pointer tracking.

3.1.7.4 Pointer Obfuscation

Pointer tracking in RUNTIMEASLR can generally handle pointer obfuscations, as encryption and decryption are symmetric, i.e., the encrypted pointers will be recognized when they get decrypted. However, we encounter a problem when pointer patching happens on encrypted or otherwise obfuscated pointers. If we do not know how the pointers are encrypted, it is impossible to patch the encrypted pointers in memory; therefore, we assume all pointer

obfuscations applied in the protected program must be known to RUNTIMEASLR. To better handle this, we provide a detection to help users identify pointer obfuscations. Note that, in the case of Nginx, we only found one such case – `glibc`'s mangled pointers.

3.1.8 Related Work

3.1.8.1 Pointer Tracking

The heart of runtime re-randomization is accurately tracking all pointers. Pointer tracking has been well-studied for object bounds checking for memory safety, and pointer protection. Existing works either employ type-based analysis or heuristics-based analysis to identify pointers. Type-based analysis [88, 98, 131, 133] statically infers the type information of an object (e.g., a pointer). It is efficient and easy to use; however, it suffers from a high number of false negatives. Pointers with non-pointer type (e.g., base addresses) and the ones prepared by the OS are not covered. Memcheck [163], Clause et al. [40], and Raksha [53] empirically specify the pointer propagation rules to track pointers. Although they specified very detailed rules, it is still difficult to cover all cases, due to the complexity of C/C++ programs. We propose an automatic mechanism to accurately identify all pointers (§3.1.6.1). Some interesting cases that are not discussed in heuristics-based analysis papers are shown in Table 13.

3.1.8.2 Re-randomization

Morula [102] is one of most related works, as it also targets the ASLR limitation with `fork`. Android adopts the Zygote model that starts app by forking, so that every app shares the same address space layout. Morula addresses this problem by simply maintaining a pool of pre-forked but re-randomized Zygote processes (template), so that when an app is about to start, one Zygote process is picked, and it starts execution from its entry point. Unfortunately, in our case of daemon processes, its child processes are forked on the fly, which starts execution from the point after `fork()` rather than the entry point. The semantic-preserving requirement of re-randomization makes RUNTIMEASLR fundamentally more difficult than

Morula. We cannot simply `fork-exec` the child process; rather, have to perform runtime re-randomization. Similarly, ASR [71] performs re-randomization at load-time, so not all semantics are preserved. Isomeron [54] and Dynamic Software Diversity [46] dynamically choose targets for indirect branches at runtime to reduce the probability of predicting the correct runtime address of ROP gadgets. In their proposals, clone-probing attacks are still effective, as child processes still share the same memory layout. TASR [19] re-randomizes code sections after a pair of socket read/write. It requires compiler support and cannot protect data pointers.

3.1.8.3 Code and Data Diversification

Code diversification [101, 142] makes multiple copies of the code, which preserves the semantics. An assumption of Code diversification is that the attacker cannot get the same copy of the code. This assumption does not hold when there are memory *overread* vulnerabilities in the program [168]. Data layout randomization [18, 37, 107] mitigates buffer overrun by making the offset between the target data (e.g., return address) and overflow point unpredictable. However, clone-probing attacks can still work under code or data diversification, since the child processes still share their parent's layout.

3.1.8.4 Fine-grained ASLR and Control-flow Integrity

Fine-grained ASLR [12, 56, 71, 80, 92, 187] and Control-flow Integrity [3, 55, 123, 127, 136, 193, 194] aim to make the exploit more difficult after ASLR is bypassed. Complementarily, RUNTIMEASLR aims to defend against the first step– bypassing ASLR. Therefore, RUNTIMEASLR is orthogonal to these techniques.

CHAPTER IV

PROTECTING SENSITIVE DATA FROM INFORMATION LEAKS

Eliminating information-leak vulnerabilities and fixing insecure designs address root causes of information leaks. However, in practice, we are not able to completely fix these vulnerabilities or insecure designs because of the complexity of systems and the continuously introduced system features. For example, we cannot even identify all side channels that may leak information, thus cannot prevent all information leaks. As a response to this problem, we instead strive to protect certain sensitive data, even in the presence of information leaks. Such protection is orthogonal to fixing root causes of information leaks; it acts as the second line of defense against information leaks. Towards this goal, we have developed ASLR-Guard, which prevents code-pointer leaks to defeat code-reuse attacks [164], and BUDDY, which detects memory disclosures with replicated execution.

4.1 ASLR-GUARD*: Stopping Code-Pointer Leaks*

Since W⊕X became a de-facto security mechanism in modern operating systems, code injection is no longer a viable general attack technique. In particular, attackers can no longer place arbitrary shell code in the target process' *data* space and hijack control to start executing that shell code. Instead, attackers now rely on code reuse attack that hijacks the control flow to *existing code* in the target process (in an unintended manner). Ever since their first introduction, code reuse attacks have evolved from simply jumping to some sensitive library functions (a.k.a. return-to-libc) to chaining up small snippets of existing code (a.k.a. gadgets) with mainly returns and indirect calls/jumps to allow the attacker to perform arbitrary computations [165]. However, there are two prerequisites for these attacks: (1) a prior knowledge of the location of existing code gadgets, and (2) the ability to overwrite some control data (e.g. return addresses, function pointers) with the correct values

so as to hijack the control flow to the targeted gadgets.

Address Space Layout Randomization (ASLR) aims to make the first prerequisite unsatisfiable by making the addresses of code gadgets unpredictable. In theory, ASLR can complement W$\oplus$X and stop code reuse attacks effectively. By randomizing the location of different code modules, and even various instructions within code modules [80, 101, 142, 187], attackers without *a priori* knowledge of how the randomization is done will not know the location of the code that they want executed. As a result, attempts to launch code reuse attacks will likely direct the hijacked control to a wrong location and cause the attack to fail (usually with a program crash). It has been widely used in modern operating systems, and proven to be extremely efficient. However, research in recent years has shown that many implementations of ASLR fail to live up to its promises of stopping code reuse attacks [20, 65, 161, 166, 174]. In some cases, the ASLR implementations fail to provide enough entropy thus are subject to brute-force attacks [20, 65, 166]. A more serious and fundamental problem for randomization based protection mechanisms is information-leak vulnerabilities [161, 174], which breaks the implicit assumption that attackers cannot learn the randomness. With information leak, the attacker usually can obtain a "post-randomization" pointer to the location of a known module (e.g. specific function in a particular library) [20, 65, 166]. For ASLR that works at a module-level granularity (i.e., it does not modify the relative location of different instructions within a module), this kind of leak will allow an attacker to identify all target addresses within that module. Even for instruction-level randomization, with a sufficiently powerful information-leak vulnerability, attackers can exploit a single code address leak and repeatedly read code pages to eventually locate all the code gadgets necessary to launch even the most sophisticated code reuse attacks [168]. More devastating, information leak is quite prevalent and the number of instances discovered is increasing in general, as shown in Figure 25.

One way to solve this problem is to completely prevent code address leak. However,

[1] http://www.cvedetails.com/vulnerabilities-by-types.php

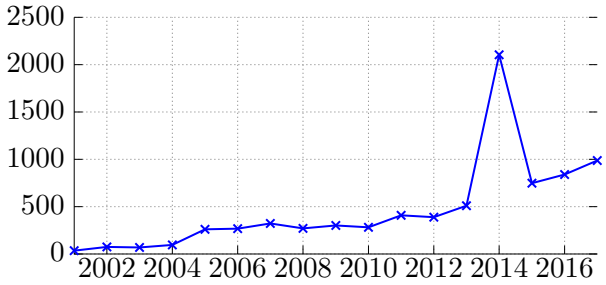

Figure 25: The number of information-leak vulnerabilities reported by CVE Details[1]. It is still increasing in general.

detecting or preventing code address leak is a very challenging problem. Due to the complexity of programs written in low level programming languages, it is even not clear about if one can find all pointers or data that can be leveraged to infer the code address. Moreover, code pointer and data can be inter-casted during the data propagation. And some code pointers (e.g., return addresses) are frequently dereferenced. Therefore, it is very challenging to correctly and efficiently protect code pointers.

In this work, we propose ASLR-GUARD, a system that completely prevents code address leak to stop code reuse attack. The main idea is to either prevent code pointer leak or render any leaked code pointer useless in the construction of code reuse attack, so that we can reclaim the benefits of ASLR and prevent code reuse attacks. In particular, we propose three techniques: (1) completely decouple code and data by remapping all code to a random address, so a data pointer cannot be used to infer the location of code; (2) store all sensitive code locators (Hereafter, we use the term **code locator** to refer to *any pointer or data that can be used to infer code address*) in a secure storage, and (3) whenever a *code locator* is going to be propagated to the regular memory regions, we encrypt it to prevent attackers from exploiting the value of this code locator or using this code locator to hijack the control flow to arbitrary addresses.

We have implemented a prototype system ASLR-GUARD, which consists of two main components, a modified compilation toolchain that includes compiler, assembler and linker; and a modified dynamic loader. The first component is responsible for instrumenting an

input program (in source code) and all its libraries so that all code locators can be identified and protected by our policies. The runtime component allows us to perform necessary initialization of our environment. Our evaluation results show that (1) ASLR-GUARD can thwart all the code locator leaks we have tested, even if attackers can dump all data regions; (2) with the capability of resisting against code locator leak attacks, ASLR-GUARD can reinforce ASLR to stop advanced code reuse attacks [20]; and (3) it incurs almost no runtime overhead (< 1%) on the SPEC benchmark.

In summary, the contributions of this work are:

- We perform a systematic analysis to identify all sources of sensitive code locators, and implemented a memory checking tool to verify that we did identify and protect all code locators.

- We propose a hybrid approach that employs isolation to protect most code locators (for better performance and security), and encrypt the remaining code locators that are hard to isolate.

- We have implemented a prototype system, ASLR-GUARD, and our evaluation shows that ASLR-GUARD incurs a negligible runtime overhead on the SPEC benchmark and left no single code locator unprotected.

A point worth noting is that protecting pointer by encryption is a general approach, but the challenge is how to achieve both security and efficiency. For example, PointGuard [43] sacrifices security for performance by XORing all pointers using a single key. Such scheme is not secure against chosen-plaintext attacks. Further, PointGuard is also vulnerable to forgery attacks due to missing integrity checks. AG-RandMap overcomes these problems by proposing a novel and efficient encryption scheme (Figure 27). Furthermore, as we will show in §4.1.2, our work overcomes a major technical challenge of identifying all the code locators that need to be encrypted. Finally, our policy for handling code locators on

stack allows us to improve both the security and, very significantly, the performance of ASLR-GUARD.

4.1.1 Threat Model

To make sure our solution is practical, we define our threat model based on strong attack assumptions, which are commonly used in projects related to code reuse attacks [20, 65, 166, 168]. As the trusted computing base (TCB), we assume a commodity operating system (e.g., Linux) with standard defense mechanisms, such as non-executable stack and heap, and ASLR. We assume attackers are remote, so they do not have physical access to the target system, nor do they have prior control over other programs before a successful exploit. We assume the platform uses 64-bit virtual address space, as 32-bit address space cannot provide enough entropy and 64-bit processors are widely available.

For the target program, we assume it is distributed through an open channel (e.g., Internet), so attackers can have the same copy as we do. This allows them to do any analysis on the program, such as finding vulnerabilities and recomputing all possible code gadgets. We assume the target program has one or more vulnerabilities, including control-flow hijacking or information-leak vulnerability, or both kinds at the same time. More specifically, we assume the program contains at least one vulnerability that can be exploited to grant attackers the capability to read from and write to an arbitrary memory address. We further assume this vulnerability can be exploited repeatedly without crashing the target program, so attackers can use the *arbitrary memory read/write* capability at will. Since arbitrary memory read/write on the 64-bit virtual address space is probabilistically impossible to achieve attacker's goals, we assume all arbitrary memory read/write will be based off on the memory addresses leaked from the previous vulnerability (or second-order guess based on the leaked code locator). Finally, we assume the ultimate goal of the attackers is to divert the control flow and execute arbitrary logic of their choice.

Although there are a few known explicit leak channels (e.g., `/proc/self/maps`), many

Table 17: A summary of all code locators that can be used to infer code address. All of them are protected by ASLR-GUARD

Category	Type	Applied protection
Load-time locators	**L1**. Base address	Encryption + Isolation
	L2. GOTPLT entry	Isolation
	L3. Static/global pointer	Encryption + Isolation
	L4. Virtual function pointer	Encryption + Isolation
Runtime locators	**R1**. Return address	Isolation
	R2. GetPC	Encryption
	R3. GetRet	Encryption
OS-injected locators	**O1**. Entry point	Encryption
	O2. Signal/Exception handler	Encryption
Correlated locators	**C1**. Jump table entry	Isolation
	C2. Data pointer	Decoupling data/code sections

security enhanced Linux distributions such as PaX disable `/proc`-based pointer or layout leaks. We assume there is no explicit leak through the platform itself.

Out-of-scope threats. Since we assume the OS as our TCB, we do not consider any attack that tries to exploit an OS vulnerability to gain control over the target program. Given our threat model, we focus on preventing the attacks that exploit vulnerabilities to bypass ASLR, and then overwrite control data to hijack control-flow. Non-control-data attacks [36] (e.g., hijacking credential data or metadata of code locators, like object pointers) are out of our scope. We also do not consider information exfiltration attacks.

4.1.2 Code Locator Demystified

Since many kinds of data besides code pointer can be leveraged to infer the address of code, such as the base address of `text` section and offset in a jump table, we use the term *code locator* to refer to any pointer or data that can be leveraged to infer code address. Although many previous works like the JIT-ROP attack [168] have shown that a single code locator leak may compromise the whole ASLR protection, to the best of our knowledge, there was no existing work that systematically discussed the composition of code locators. In order to provide a comprehensive protection against information leak, we first conduct a

systematic analysis to discover all code locators in a program, and discuss the completeness of coverage.

4.1.2.1 Discovery Methodology

When a program is starting, the kernel is responsible for creating the address space of the process and loading the ELF binary and dynamic linker, while the relocation of code pointers is left for dynamic linker. Dynamic linker then loads all required libraries and performs necessary relocations before the loaded code is executed. Once all relocations are done, the dynamic linker transfers the control to the program's entry point. During execution, the program may (indirectly) call function pointers, or interact with OS, e.g., signal handling. We categorize code locators mainly based on the life cycle of the program execution. Then we try to identify all code locators at different program execution stages. More specifically, we first thoroughly checked the source code of dynamic linker (`ld.so`), libc libraries, and gcc to understand how code locators could be generated at load-time and runtime of the program execution. As for code locators that are injected by the OS into the process' address space, we implemented a memory analysis tool to exhaustively check if *any 8-byte* in readable memory points to any executable memory in the target program or bases of modules, and we performed this check before and after executing every system call. The memory analysis tool also serves to validate our process for discovering code locators injected by the dynamic linker and the program itself (e.g., checking the memory at the entry/exit points). Once our memory analysis tool discovers new kinds of code locators, we manually verify and categorize them. We then append the new "category" to the identified code locator set, and run memory analysis tool again to find more kinds of code locators. The iterative process ends until no new code locators are reported. We summarize our results of such exhaustive memory analysis in Table 17.

4.1.2.2 Code Locators at Different Stages

Load-time Code Locators. During load-time, there could be various kinds of code locators computed and stored in the memory, e.g., static function pointer and `.gotplt` entry, as shown in §4.1.2. Although it is hard to iterate all code locators produced at load-time, we found the fact that, suppose ASLR is fully enabled, all code locators must be relocated before being dereferenced. With this fact in mind, we designed a general approach to discover and protect `all` load-time code locators. We control the relocation functions in dynamic linker, so that all load-time code locators must go through our relocation, and therefore we can enforce our protections for them, e.g., isolation and encryption.

Runtime Code Locators. Code locators can be generated at runtime by deriving from the program counter (e.g., `RIP`). In x86_64 platform, the `RIP` value can be loaded either by a `call` instruction or a `lea` instruction. `call` instruction pushes return address on the top of stack, and when the callee returns, the return address is dereferenced. Usually, people believe `call` and `ret` are paired. In our analysis, we indeed found some corner cases in which return address is not used by `ret`. As the code locator type **R3** shown in §4.1.2, `setjmp` loads return address using a `mov` instruction, and the return address is then dereferenced as a function pointer by `longjmp`. Besides the return address, runtime code locators can also be generated by `GetPC` (e.g., `lea _offset(%rip), reg`). For example, if there is a function pointer that is assigned with an address of a local function, typically `GetPC` will be used to get the function address. Since we have the complete control of the compilation toolchain, we could properly discover `all` runtime code locators (e.g., the ones introduced by `call` or `GetPC`) and enforce the corresponding protections, which are elaborated in §4.1.3.

OS-injected and Correlated Locators. As ASLR-GUARD is designed to work with unmodified commodity kernel, how the program interacts with kernel needs be analyzed in order to discover code locators that might be injected or transmitted by the kernel. We used our memory analysis tool to check the memory right before and after a syscall. Based on

our analysis, we found that the program entry pointer can be stored on the stack by kernel. Also kernel may save the execution environment, including `RIP` value, for signal/exception handling. Since data sections and code sections are mapped together, the offsets between them are fixed. Data pointers can be leveraged to infer code addresses. We call such pointers the correlated code locators, which we correctly handle to prevent intentional code address leaks.

4.1.2.3 Completeness of the Analysis

In an ASLR-enabled program, all static code locators (e.g., the ones in `.gotplt` and virtual table) must be relocated at load-time. As we can control the relocation routine, we can guarantee to cover all such code locators. Another source of code locator at load-time is that a code locator is calculated based on a base address. So we also collect all cases in dynamic linker that access the base address stored in `link_map`. In this way, we can cover all load-time code locators. Note that we will remap all code sections to random addresses, any missing code locator at load-time (although we did not meet such case in our evaluation) will crash the program, which means our analysis for load-time code locators is "fail-safe". For runtime code locators, as we have the control of whole toolchain, we can easily guarantee to find all runtime code locators (i.e., we catch all instructions that access return address or `rip`).

As for code locators that are injected by the OS, we take a dynamic analysis approach. While this dynamic analysis approach is not necessarily complete in theory, we will argue that in practice the way the OS setups a process' address space (and thus all the bookkeeping code locators it may inject into various user space data structure) should be the same for every process; a similar argument should apply for any code locator injected for signal handling.

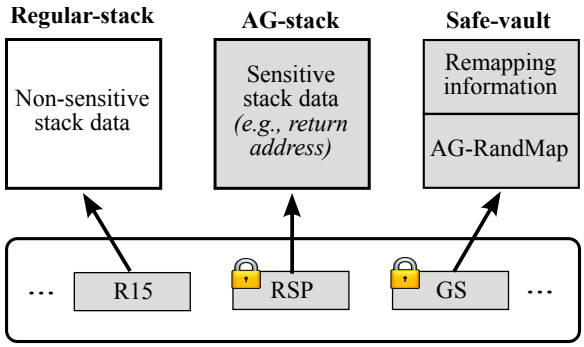

Figure 26: Register usage semantics in ASLR-GUARD. The usage of `rsp` and `gs` will be securely regulated by ASLR-GUARD's compiler, so never be copied to the program's memory, nor accessed by the program's code.

4.1.3 Design

In this section, we present the design of ASLR-GUARD. Based on previous discussion and existing attacks [20, 153, 168], we know that as long as attackers can exploit information-leak vulnerabilities to obtain a valid code locator, ASLR will continue to be nullified, no matter how much we improve the granularity of the randomization. So the goal of ASLR-GUARD is to: *completely prevent leaks of code locators*.

Figure 26 demonstrates the high-level idea of ASLR-GUARD. First, it decouples code sections and data sections. By doing so, the process' address space is virtually divided into two worlds: the data world and the code world. This step eliminates all the implicit relationship between data and code (C1). Second, ASLR-GUARD acts as the gateway between these two worlds: whenever a code address is about to be leaked to the data world, ASLR-GUARD encrypts the address as a token; and whenever a token is used to perform *control transfer*, ASLR-GUARD decrypts it. By doing so, ASLR-GUARD eliminates all explicit code locator leaks. To implement this scheme, we developed four key techniques: *safe vault*, *section-level randomization*, *AG-RandMap* and *AG-Stack*.

4.1.3.1 Safe Vault

Safe vault is an "isolated" memory region where all the plaintext code locators (except `R1`) listed in Table 17 are stored. We divide the safe vault into two parts. The first part is for

storing the information of all remapped sections, which includes the base address for each section before and after the remapping, as well as the size of each section. The second part is the random mapping table used for code locator encryption/decryption.

In ASLR-GUARD, we guarantee the isolation of the safe vault with the following design. First, the base address of the safe vault is randomized via ASLR. Second, the based address is stored in a dedicated register that is never used by the program [2], and its content is never saved to the data memory. As a result, one can only locate the safe vault by brute-forcingly guessing its base address, which is prohibited by ASLR on 64-bit systems.

4.1.3.2 Section-level randomization

The purpose of the section-level randomization is two-folded: first, we use it to stop attackers from deducing the location of our code using leaked data pointer; second, we would like to further protect data structures that contain code locators (namely jump tables, `.gotplt` tables and virtual function tables) without paying the price of encrypting all the code locators in them.

For our first goal, we decouple code from data by remapping all code sections (code sections can be identified by looking up ELF header) to random addresses. Since the offset between code and data is changed during this process, we further adjust the constant offsets in all data-access instructions. This is done at load time based on the relocation information generated by our linker. For our second goal, we modified the compiler to make sure that: (1) jump tables are always emitted in `rodata` sections, (2) addresses pointing to, or derived from the content of, code locator tables like jump tables and `.gotplt` tables always stay in registers and are never saved to the memory. With these two guarantees, we can protect code locators in these data structures by simply remapping them to random addresses, without ever needing to encrypt the code locators stored in these tables. One exception to our second rule is the virtual function tables. Since we cannot guarantee that virtual function pointers

[2] On x84-64 platform, the segmentation register `gs` is the most suitable candidate for this purpose.

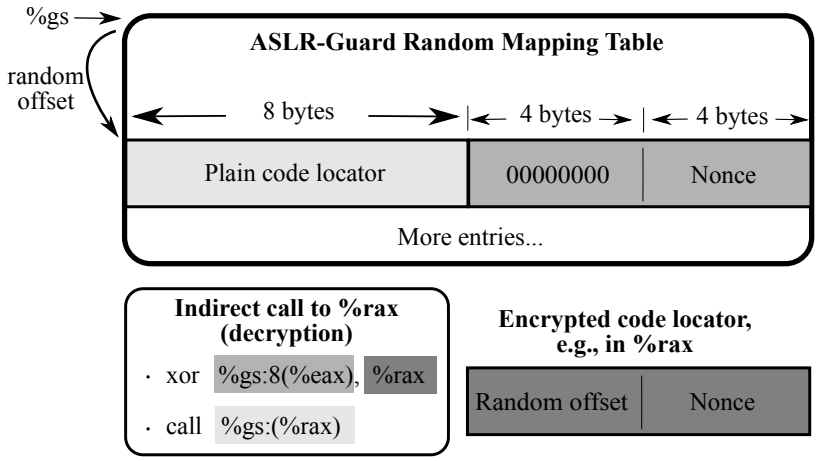

Figure 27: Code locator encryption/decryption scheme.

will never be stored to the unsafe data world, we encrypt virtual function pointers after the virtual tables are remapped to a random address. After the remapping, we use the safe vault to store the remapping information for each of the protected section so we can later patch up accesses to them. One final point to note is that to make sure the remapped location of the various tables are truly random, we wrapped the *mmap()* function to provide it random base addresses. As the primary goal of ASLR-GUARD is to harden ASLR, instead of to improve the entropy of ASLR, we set the same entropy as in the default 64-bit Linux ASLR (28-bit). We acquire the randomness from cryptographic secure source, i.e., the `RdRand` instruction on modern Intel processors. There are alternatives from `/dev/random` that could be used.

4.1.3.3 AG-RandMap–Locator Encryption

In this subsection, we define our encryption scheme for encrypting code locators. This encryption scheme is designed to achieve two goals. First, it must be strong enough to stop code reuse attacks even when some (or all) of the encrypted code locators are leaked. We also assume that attackers can know what functions the leaked code locators point to. In order to achieve this goal, our scheme need to prevent attackers from efficiently deriving valid encrypted code locators that will be dereferenced to direct the control flow to code gadgets. In particular, our encryption scheme should make the task of deriving a valid encrypted code locator as difficult as brute-forcingly guessing the gadgets under ASLR. The

second requirement for our encryption scheme is performance, especially for the decryption operation. This is because code locator generation is a relatively rare operation, but code locator dereferencing (e.g., indirect call) is much more frequent.

To achieve these goals, we propose AG-RandMap, an efficient `random mapping table` mechanism, as our encryption scheme. The main idea of AG-RandMap is illustrated in Figure 27. To encrypt a code locator, we will randomly pick an unused 16-byte entry in the mapping table. The first 8 bytes will be used to store the plain code locator being encrypted; the next 4-bytes is zero that is used to get the random offset with XORing during decryption, and the last 4-bytes contains a random nonce (with 32-bit entropy) for integrity check. As in 4.1.3.2, we acquire the randomness from cryptographic secure source (i.e., the `RdRand` instruction provided by modern Intel processors). After building the new table entry, the 4-byte random offset from the base of the mapping table to the new entry is concatenated with the nonce, and returned as the encrypted code locator. Note that AG-RandMap does not use any key for encryption, but generates an entry in the mapping table and saves the encrypted code locator in the corresponding register or memory using "one-time" random value.

Decryption under our scheme is performed very efficiently and involves only one extra `xor` operation on top of the original indirect control transfer. The code locator decryption process is also illustrated in Figure 27. Assuming the encrypted code locator is in `%rax`, we first use `%eax`-plus-8 as an offset to read 4-bytes zero and the nonce from the table, and `xor` it with the encrypted code locator. If the nonce is correct, the result will be a valid offset within the mapping table, which then decrypts `%gs:(%rax)` to the plain code locator. Otherwise, the offset will be outside of the table, transferring the control to a random address that is highly likely to be either unmapped or not executable, thus crashing the program. In other words, for an attacker to forge a valid encrypted code locator that will direct the control flow to a plaintext address stored in our AG-RandMap, he will need to know the correct nonce to use for the target table entry. To do so, he can either guess with a successful chance of 2^{-32};

or try to locate our AG-RandMap, which is equivalent to brute-forcing ASLR. Either way, our encryption scheme satisfies the first requirement.

C++ support. To support polymorphism, C++ uses a virtual function table (vtable) to store the pointers of virtual functions for each class with virtual functions. And in each object of class with virtual function, vtable pointer(s) (vptr) are used to point to the right vtable. ASLR-GUARD generally protects the function pointers inside vtable by encryption. However, as vtable pointers are stored in objects in unsafe memory, leaking vptr may help attackers know all encrypted virtual function pointers and allow COOP attack [153]. To prevent such attack, ASLR-GUARD further encrypts vtable pointers so that attack cannot even know the locations of encrypted virtual function pointers, which is an essential step for COOP attack[153].

4.1.3.4 AG-Stack

As discussed in §4.1.2, some dynamic code locators are stored on the stack, such as return addresses (R1) and OS injected code locators (O2). To prevent leaking return address through stack, one approach is to apply the same encryption/decryption scheme in §4.1.3.3. However, this approach is not suitable for protecting return address on the stack. First of all, the return addresses are generated very frequently, and our encryption scheme does not support frequent encryption very well. Second, it will require extensive modification to support encryption over OS injected code locators. For these reasons, we propose an efficient alternative solution, namely AG-Stack to protect R1 and O2.

Our approach is similar to the traditional shadow-stack techniques [68, 149] that are used to prevent return addresses from being modified by attacks. However, they protect return addresses by costly instrumenting `call/ret` pairs. For example a single `ret` is instrumented with at least 4 instructions that load return address in the shadow stack, adjust the stack, check equivalence, and `ret`. Therefore, traditional shadow-stack approaches face three problems: (1) `call/ret` are frequently executed by CPU, instrumenting them incurs

significant performance overhead; (2) cases where `call/ret` don't come in matching pairs, e.g., `GetPC`, are difficult to handle; (3) "return addresses" injected by the OS will not be automatically protected. To address these issues with traditional shadow-stack techniques, we propose a novel mechanism, namely AG-Stack.

Besides being much more efficient, AG-Stack has several advantages: (1) it provides the capability to store more sensitive data; and (2) it does not change the semantics of original code, so special cases like unpaired `call/ret` instructions and signal handling can be naturally supported. The high level idea of AG-Stack is to maintain two stacks by re-arranging the usages of two registers. Specifically, we use the ordinary stack (i.e., the one accessed through `RSP`, as shown in Figure 26) as our AG-Stack to store sensitive on-stack data like return addresses and function pointers used for exception handling. All other data, like stack variables, register spilling and function call arguments are stored on a regular stack that accessed through `R15`, as shown in Figure 26. The security of AG-Stack is guaranteed in a way similar to safe vault. As in safe vault, its base address is randomized. Also, since AG-Stack does not store any program data, there is no data pointer pointing to it, so attackers cannot derive its address through memory traversal. Finally, whenever the stack pointer is stored in memory (e.g., during `setjmp`), we will encrypt it using the scheme in §4.1.3.3. As such, with AG-Stack, we achieve a better performance but a guaranteed security for all code locators saved on the stack.

AG-Stack is very similar to the safe-stack from CPI [98]. Both impose no runtime overhead. The difference is that safe-stack performs type-based analysis and intra-procedure analysis to find unsafe stack objects. As shown in Section §4.1.2, type-based analysis may not cover some non-pointer code locators. AG-Stack eliminates these analyses by thoroughly re-arranging the register usages.

Multi-thread support. In Linux, multi-thread feature is enabled with Native POSIX Thread Library (NPTL), which creates a new thread with the `clone` system call. Specifically, to create new thread, NPTL first allocates a new stack and stores thread descriptor at the end

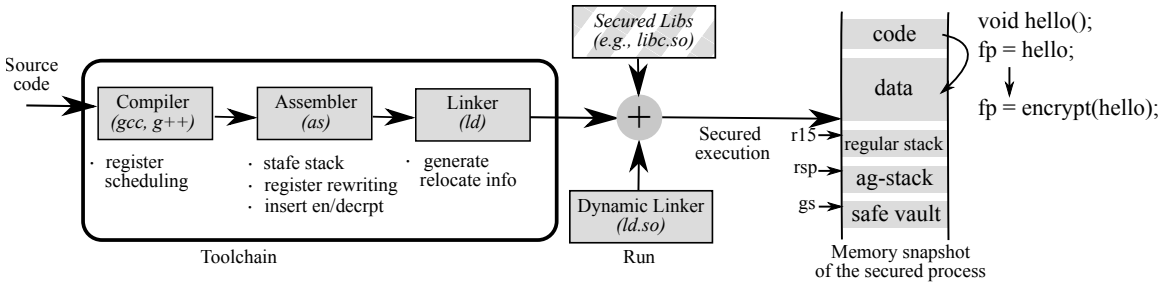

Figure 28: An overview ASLR-GUARD's architecture. It has two components: toolchain (e.g., compiler) and runtime (e.g., loader and libc). Code locators will never be copied into the program's data memory, avoiding potential leakages.

of the new stack. And then it saves the thread function pointer and its arguments in the new stack. After that, `clone` system call switches the context, and the thread function is called. Similarly, to support multi-thread in ASLR-GUARD, for each new thread, we allocate a regular stack (with the same size) whenever the new (safe) stack is allocated, and release the regular stack when the thread exits. The regular stack information (e.g., stack address and size) is saved in thread descriptor that is stored in AG-Stack.

4.1.4 ASLR-GUARD Toolchain

In this section, we describe our prototype implementation of our protection scheme. We first give an overview of our prototype, then we discuss the detail of each component of ASLR-GUARD.

4.1.4.1 Architecture

Figure 28 shows the overview of ASLR-GUARD's architecture. To remove all code locator leaks, ASLR-GUARD requires instrumenting all loaded modules, including modules of the target program as well as all shared libraries. In this work, we assume the access to the source code of the target program is available and the instrumentations are done through re-compilation.

ASLR-GUARD consists of two major components: the *static toolchain* and the *runtime*. The static toolchain is in charge of (1) implementing the AG-Stack, which will be used to securely store return address and OS injected code locators; and (2) instrumenting the

Component	Tool	Lines of code
Compiler	gcc-4.8.2	120 lines of C
Assembler	as-2.24	900 lines of C
Linker	ld-2.24	180 lines of C
Dynamic Linker	eglibc-2.19	1,200 lines of C
Memory analyzer		800 lines of Python
	Total	3,200 lines of code

Figure 29: Toolchain modification made for ASLR-GUARD.

program to correctly encrypt runtime code locators and decrypt all code locators. The ASLR-GUARD runtime contains three majors parts. The first part is the dynamic loader, which is in charge of (1) initializing the AG-Stack and the safe vault, (2) decoupling code sections from data sections, and (3) encrypting all load-time code locators. The second part is the standard `C/C++` libraries that will be linked into most programs. Among these libraries, some need special care because they need to handle some OS-injected code locators in special ways (e.g., `setjmp`).

We implemented our prototype based on the GNU toolchain: `gcc`, `binutils` and `glibc`. Figure 29 summarizes the our efforts of modifying the GNU toolchain.

4.1.4.2 Compiler

We modified the GNU compiler `gcc` to assist implementing the code locators encryption and AG-Stack. For code locator encryption, we let `gcc` insert a flag for data-read instructions of global data pointers to differentiate global function pointers in the generated assembly code. For AG-Stack part, we performed several modifications in `gcc`. First, we need to reserve a register for the unsafe stack. In `gcc`, this can be done through the `-ffixed` option. But a more important question is, which register should be chosen. In ASLR-GUARD, we choose this register using two criteria: (1) it should be one of the least used general registers in handwritten assembly code, and (2) it should be one of the callee-saved registers (i.e., non-volatile). The first requirement is for that compiler can only guarantee that the code generated by them does not use the reserved register, however, any handwritten assembly (including inline assembly) may still use this register. They need to be modified to use

other registers (e.g., we found and modified about 20 cases in `glibc`). So choosing the less frequently used register helps us minimize the modifications. The second requirement is for compatibility with legacy uninstrumented binary. Because we reserved the `RSP` for AG-Stack, by using a callee-save register, it ensures that both code can work correctly. Based on these criteria, we chose `R15` as the register for the regular stack on our target platform.

Second, we want to make sure that besides `call/ret`, there is no implicit `RSP` modifications. On x86-64, such modifications can be introduced by instructions, `push/pop`, `enter/leave` and `call/ret` [83, Ch. 6.2]. `Push/pop` are for saving/restoring the target registers, and `enter/leave` is a shortcut for saving/restoring the frame register. Our elimination of `push/pop` is done in two steps. We first leveraged existing compiler optimizations to reduce the usage of `push/pop`. In particular, modern compilers like `gcc` in most scenarios prefer using `mov` instruction for performance gain, as CPU can do pipeline scheduling for `mov` instructions. So we modified `gcc` to always prefer using `mov` instruction for passing arguments (as if `-maccumulate-outgoing-args` and `-mno-push-args` are always set) and saving/restoring registers at function prologue/epilogue. For the remaining `push/pop` operations, our assembler will replace them with corresponding `mov` operations on `R15`. The elimination of `enter/leave` is done similarly

Finally, we need to re-align the stack. The System V AMD64 ABI uses `XMM` registers for passing floating point arguments. And reading/writing these registers from/to memory requires the memory address to be 16-byte aligned. As we split the stack into AG-Stack and regular stack, the regular stack needs to be re-aligned. We enforced the alignment by leveraging `gcc`'s supports for multiple platforms. Specifically, on platforms like ARM, function invocations do not automatically save the return address on stack, so `gcc` already understands how to align stack when there is no return address on stack. Using this support, we modified `gcc` to treat our target platform as if it does not explicitly save the return address on stack (by setting `INCOMING_FRAME_SP_OFFSET = 0`).

4.1.4.3 Assembler

We modified the GNU assembler `gas` to perform code instrumentations for code locator encryption/decryption and AG-Stack. The first task is to encrypt every runtime code locator (type R2, R3 in Table 17). For position-independent ELF executables, runtime code locators are created through retrieving the `PC` value (e.g., `mov (%rsp), %rax` may load return address) or a PC-relative address (e.g., `lea offset(%rip), %rax`). ASLR-GUARD uses these signatures to identify all potential code locators. Specifically, we perform a simple intra-procedure analysis (check if `RSP` points to return address) to find all instructions that load return addresses. And we find the instructions of loading "PC-relative" code locators based on the metadata (e.g., indicating if a symbol is a function) contained in the assembly files and the global data pointer flag inserted by our compiler 4.1.4.2. Note that, for handwritten assembly, there is no easy way to distinguish the ones for global data access and global code access at assemble time. To address this issue, our assembler first conservatively instruments both of them, and defer it to our linker to remove the false positive ones, as linker has code boundary information to verify code locators. For each identified instruction, our modified assembler will insert an encryption routine to immediately encrypt the code locator, as shown in Figure 27

Next, we instrument the target program to correctly decrypt code locators. Specifically, we first instrument every indirect call with code address in register in the ways shown in Table 27. If the target is stored in memory, we first save `RAX` and load the target into `RAX`. Then we perform the decryption as the one in Table 27. One difference is that, to recover `RAX`, we temporarily store the decrypted code address in AG-Stack, do recovery, and call into the code pointer in AG-Stack. Indirect jump for invoking function is instrumented in the same way.

The third task is to replace all `push/pop` and `enter/leave` operations with explicit stack pointer operations. After this, ASLR-GUARD will replace all `RSP` occurrences with `R15`, except when `RSP` is used to access return address in handwritten assembly. Using the

same intra-procedure analysis mentioned above, we can find all such cases and skip the replacements.

4.1.4.4 Static Linker

We modified the GNU static linker `ld` to perform two tasks. The first task is to provide relocation information to the dynamic linker. This is necessary because once we decoupled code sections from data sections, existing data access instructions will no longer work because the offsets have changed. To fix this problem, ASLR-GUARD creates a new relocation table similar to the `.rela.dyn` to provide this information, i.e., which 4-bytes need to be patched. The second task is to remove encryptions for global data access in handwritten assembly, conservatively added by assembler. This task is performed here because the linker has the information of all static modules, so it can identify whether an instruction is for accessing data or code.

4.1.4.5 Dynamic Linker

We modified the dynamic linker `ld.so` of `glibc` to initialize the runtime, decouple code sections from data sections, and encrypt load-time code locators. When running an executable, Linux will first load its interpreter (i.e., the dynamic linker), jump to the entry of its entry and let the interpreter handle the target binary. This means the dynamic linker is the very first module to be executed. For this reason, we insert our runtime initialization code at the entry of the dynamic linker (`_dl_start()`). The regular stack is initialized by allocating a new memory region with a random base, and copying all the arguments and environment variables from the initial stack. Safe vault is setup by allocating another random memory region and assigning the `gs` segment register to its base address. Once all the initializations are done, the control is transferred to the remapped code of dynamic linker, and the original one is unmapped.

The decoupling is done by remapping the corresponding sections to random addresses as

the way mentioned in Section 4.1.3.2. After remapping, ASLR-GUARD will encrypt all load-time code locators (see Table 17). This is done during the relocation patching. Specifically, after randomizing the base address, the original dynamic linker already provides some relocation functions to patch all those load-time code locators with correct code addresses. In ASLR-GUARD, with the help of this feature, we modify these relocation functions to patch the load-time code locators with encrypted values, instead of patching them with the real code addresses.

4.1.4.6 Standard C/C++ Library

Protecting the standard C/C++ libraries is very important because they are linked into every `C/C++` program. Yet they contain hand-written assembly that may need special care, to handle some corner cases may have compatibility problems with the automated hardening techniques. Unfortunately, many previous work did not show enough discussions on handling these libraries. In this subsection, we try to shed some light on this topic.

We first compile `eglibc-2.19` and `gcc-4.8.2` with our own toolchain. Then we handle three main cases in handwritten assembly code: (1) access to return address and stack parameter; (2) stack layout mismatching; (3) indirect jump for invoking functions; In the first case, we use our intra-procedure program analysis to find instructions for loading return address (e.g., `mov (%rsp), %rax`). For each identified instruction, we insert the encryption routine right after it. However, because the regular stack does not contain the return address any longer, the offsets used to access stack parameters (x86_64 calling convention passes parameters via stack after the first 6 ones) are incorrect. We address this by manually adjusting the regular stack top at function prologue and epilogue. In total, we found 12 such cases in the handwritten assembly of `glibc`. The second case happens when the caller is in handwritten assembly and callee is in C code, or vice versa. For example, `_dl_runtime_resolve` is written in assembly, at the end of which, its local variables, return address, and parameters are released at once by a stack-pivot instruction

(i.e., `addq $72, %rsp`), as it assumes there is only a single stack. However, this is not correct as AG-Stack and regular stack are supposed to be released separately. To handle this, we manually change the assembly. In total, we handled 4 such cases in libc. As mentioned in §4.1.3.2, indirect jumps for jump table will not be instrumented, but the ones for invoking functions will be. To differentiate them in handwritten assembly, we manually verify if a jump instruction is used for function invocation or not, if yes, we instrument them to go through the code locator decryption. We found 1 case in `glibc`.

`DWARF` standard is adopted for C++ exception handler (EH). For each `throw` statement, an exception is allocated and the stack unwinding (i.e.,`unwinder`) is started. For each `catch` statement, `g++` has already generated a EH table and a cleanup table during compilation. And at runtime, the `unwinder` will walk through the stack twice to handle the exception: (1) trying to find the correct exception handler based on EH table; (2) if a matching is found, walking though the stack a second time for cleanup based on the cleanup table. To support C++ exception handling in ASLR-GUARD, we need to update these two tables and rewrite the `unwinder` to support AG-Stack. Since doing so merely requires some engineering efforts and many C++ programs do not use it (e.g., Chromium does not use it and only two SPEC benchmark programs use it), we did not support C++ exception handling in our current prototype yet, but leave it for future work.

4.1.5 Evaluation

In this section, we present the evaluation of ASLR-GUARD. Our evaluation tries to answer the following questions:

- How secure is the ASLR-GUARD's approach in theory, compared to general ASLR approach?
- How secure is the ASLR-GUARD's approach against practical memory-based attacks, empirically?

- How much overhead does ASLR-GUARD impose, in particular runtime, loading time, and space?

Experimental setup. We carried out evaluations of ASLR-GUARD to check its security enhancements and potential performance impacts. Our evaluations are mainly performed on the standard SPEC CPU2006 Benchmark suite. Beside that, we also applied ASLR-GUARD to complex and real world programs, e.g., `gcc`, `glibc` and Nginx web server. We compiled all above programs and their libraries with the toolchain of ASLR-GUARD and enforced them to use the dynamic linker of ASLR-GUARD. All programs were run on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz machine with 128 GB RAM.

4.1.5.1 Security Analysis

In this subsection, we will present our analysis to determine the probability for an attacker to hijack the control to a particular target address (let's say, `x`). For our discussion in this section, we will assume the ASLR provided by the underlying OS (in our implementation, 64-bit Linux) provides 28-bit entropy for the address of any section that has been randomized.

Referring to Table 17, we can see that if all code locators to `x` are generated from sources **L2**, **R1** or **C1** (i.e. items only reachable through `.gotplt` tables, return instructions or jump tables), the attacker can only hijack the control to `x` by overwriting the content of the safe vault. This is because ASLR-GUARD will never generate any encrypted code locator for `x`, thus attackers cannot achieve the goal by overwriting an encrypted code locator. As such, the probability for successful hijacking is the same as that of breaking ASLR, i.e. 2^{-28}, assuming the attacker knows the semantics, e.g., the offset of `x` in memory page.

On the other hand, if `x` can be reached through an encrypted code locator (i.e. `x` is the address of items in the categories **L1**, **L3**, **L4**, **R2**, **R3**, **O1**, **O2** of Table 17), and assuming an encrypted code locator for `x` exists in AG-RandMap[3], attackers have a new option by

[3]Otherwise, the situation will be the same as above, the attacker will have to overwrite values in the safe vault.

Table 18: Code locator numbers. CL: code locator.

Programs	Size of rand. map table (byte)	# all enc. CL	# load-time CL	# PLT entries	# fixed PLT entries	# enc. CL in unsafe memory	# plain CL in in unsafe memory
perlbench	14,144	884	967	131	34	603	0
bzip2	4,480	280	303	28	11	20	0
gcc	22,784	1,424	1,339	89	39	187	0
mcf	4,528	283	361	39	18	9	0
gobmk	32,720	2,045	2,104	66	26	1,771	0
hmmer	4,544	284	361	83	37	10	0
sjeng	4,592	287	365	42	21	27	0
libquantum	4,512	282	324	46	12	9	0
h264ref	4,736	296	341	64	37	16	0
astar	14,560	910	2,289	1,348	33	29	0
xalancbmk	141,696	8,856	10,275	1,449	184	258	0
milc	4,544	284	336	58	15	9	0
namd	14,880	930	2,262	1,353	36	44	0
dealII	54,784	3,424	4,884	1,473	310	81	0
soplex	21,472	1,342	2,717	1,377	254	89	0
lbm	4,528	283	316	38	13	9	0
sphinx3	4,544	284	354	76	39	10	0
Average	**21,062**	**1,316**	**1,759**	**456**	**66**	**187**	**0**

trying to overwrite some encrypted code locators residing in the unsafe memory with a value that can "decrypt" the control to `x`. For this kind of target, given a leaked encrypted code locator other than `x`, as we have argued in §4.1.3.3, and the attacker cannot read the content of AG-RandMap in safe vault, his chance of creating the correct encrypted value of `x` is 2^{-32}, guaranteed by the random nonce. Either way, the best chance of a successful attack is no more than 2^{-28}. We discuss a case of that the encrypted value of `x` is leaked in §4.1.6.

4.1.5.2 Practical Security

To see how effective ASLR-GUARD is in practice, we performed multiple empirical security analyses.

Memory snapshot analysis. As mentioned in §4.1.2, even it is unlikely, we cannot guarantee kernel will never save code locators in unsafe memory, as our current design did not instrument kernel code. To make sure there is no single plain code locator left in the unsafe memory by kernel, we did a thorough memory analysis using our memory

analysis tool described in §4.1.2. More specifically, we first applied ASLR-GUARD to SPEC benchmark programs. And then we hooked the entry/exit points of the programs, as well as all points right after syscalls, and dumped the whole memory core for memory snapshot analysis. As expected, there is no single plain code locator left in unsafe memory. All code locators are either isolated or encrypted, which prevents attackers knowing the address of any executable memory. Note that, our memory analysis is conservative and may contain false positives that some random data is falsely treated as code locator. Indeed, we found two "code locators" that point to executable memory, in `sphinx3`. However both code locators are eliminated due to they actually point to the the middle of instructions, so we believe they are false positives. Furthermore, we evaluated other numbers, e.g., the map table size, number of all encrypted code locators, number of load-time code locators, size of `.gotplt` tables, etc. Out of them, an interesting number is the one of encrypted code locators left in the unsafe memory. To get this number, we check if any 8-bytes is "pointing" to the random mapping table by decrypting it in the way shown in Figure 27. Note that such analysis is conservative, and the actual number could be even less. All the evaluated numbers are shown in Table 18. In most programs, less than 10% encrypted code locators are propagated to unsafe memory. Many of them have less than 20 ones in unsafe memory.

Nginx web server. We chose nginx to evaluate the effectiveness of ASLR-GUARD, not only because it is one of the most popular web servers but also that it has been compromised by couple of attacks [20, 65]. We took Blind Return Oriented Programming (BROP) [4] as an example to show how ASLR-GUARD defeats the return address leak, which is an essential step in BROP attack. BROP attacks `nginx 1.4.0` (64-bit) by exploiting a simple stack buffer overflow vulnerability. This attack consists of three steps: (1) guessing the return address on the stack; (2) finding enough gadgets; and (3) building exploit. In step (1), BROP proposed using stack reading approach to overwrite the return address byte-by-byte with possible guess value, until the correct one is found. Naturally, ASLR-GUARD can prevent

[4] http://www.scs.stanford.edu/brop/

Table 19: Comparison of average number of indirect branches between ASLR-GUARD and CFI works, with the SPEC benchmarks. AG: ASLR-GUARD

Programs	Original-CFI		bin-CFI/CCFIR		AG	
	ret	call	ret	call	ret	call
perlbench	1	7,722	28,066	7,722	1	603
bzip2	1	1,097	11,905	1,097	1	20
gcc	1	16,697	65,519	16,697	1	187
mcf	1	1,234	12,935	1,234	1	9
gobmk	1	7,357	22,804	7,357	1	1,771
hmmer	1	1,473	16,898	1,473	1	10
sjeng	1	1,647	12,677	1,647	1	27
libquantum	1	1,184	13,405	1,184	1	9
h264ref	1	2,707	16,191	2,707	1	16
astar	1	3,398	27,301	3,398	1	29
xalancbmk	1	15,977	133,642	15,977	1	258
milc	1	1,814	14,435	1,814	1	9
namd	1	3,413	28,162	3,413	1	44
dealII	1	6,687	120,788	6,687	1	81
soplex	1	4,513	36,512	4,513	1	89
lbm	1	1,171	12,926	1,171	1	9
sphinx3	1	1,316	15,608	1,316	1	10
AIT	**1**	**4,671**	**34,692**	**4,671**	**1**	**187**

such attack at the first step, as return addresses are not stored in regular stack at all. To verify it, we applied ASLR-GUARD to nginx, and run the exploit again. As expected, the exploit failed at guessing the return address. Moreover, we did the memory analysis during this attack. We found there were 1,474 code locators encrypted, out of which, 361 ones were propagated to unsafe memory. Again, no plain code locator was left in unsafe memory.

Average indirect targets. If we conservatively assume the attackers can read the whole unsafe memory and understand the semantics of the memory (as will be discussed in §4.1.6), attackers may reused the leaked encrypted code locators to divert control flow. Given this conservative assumption, we want to know how many encrypted code locators are left in unsafe memory. Also we define a metric `Average Indirect Targets` (AIT) to measure the average number of possible targets an indirect branch may have. The reason we do not reuse AIR (Average Indirect target Reduction) metric proposed in [194] is that even the AIR is 99.9%, the number of possible targets is still huge, especially for some large programs. As we can see in Table 19, ASLR-GUARD only left a small portion (4%) function pointers in

Table 20: Evaluation of runtime performance with only AG-Stack and full ASLR-GUARD protection, load-time overhead, and increased file size.

Programs	**Runtime**					**Load-time**		
	Orig (s)	AG-Stack (s)	AG-Stack Overhead	Full AG (s)	Full AG Overhead	Orig (μs)	AG (μs)	AG Overhead
perlbench	4.17	4.20	0.72%	4.32	3.60%	1.1	1.8	62.8%
bzip2	12.2	12.3	0.82%	12.1	-0.82%	0.8	0.9	16.6%
gcc	1.75	1.74	-0.57%	1.93	10.29%	5.5	6.5	17.9%
mcf	3.14	3.10	-1.27%	3.10	-1.27%	2.4	3.2	30.8%
gobmk	25.6	25.5	-0.39%	25.7	0.39%	3.2	5.4	67.9%
hmmer	8.07	8.06	-0.12%	8.04	-0.37%	3.6	4.1	13.8%
sjeng	5.48	5.41	-1.28%	5.56	1.46%	2.7	3.9	41.8%
libquantum	0.161	0.165	2.48%	0.165	2.48%	2.4	2.5	4.7%
h264ref	31.5	31.2	-0.95%	32.3	2.54%	3.0	3.8	25.6%
astar	15.1	14.9	-1.32%	14.5	-3.97%	3.4	4.0	17.3%
xalancbmk	0.592	0.600	1.35%	0.615	3.89%	9.9	10.2	2.9%
milc	12.6	12.0	-4.76%	12.0	-4.76%	1.0	1.6	58.9%
namd	22.0	21.1	-4.09%	21.1	-4.09%	3.5	5.2	45.9%
dealII	73.1	76.3	4.38%	73.6	0.68%	4.1	5.1	24.2%
soplex	0.070	0.071	1.43%	0.071	1.43%	3.6	5.1	42.7%
lbm	3.16	3.12	-1.27%	3.16	0.00%	2.4	3.2	35.1%
sphinx3	2.39	2.37	-0.84%	2.38	-0.42%	2.5	3.2	24.2%
Average			**-0.33%**		**0.65 %**			**31.35%(0.86μs)**

unsafe memory, which is much smaller than the ones of CFI implementations. Recently, some CFI variants make use of `type` to further reduce the AIT numbers [136, 178]. As shown in [136], the AIT number can be dramatically reduced to less than 1% compared with original one. So we believe AIT of ASLR-GUARD can also be further reduced once we consider `type` in the future.

4.1.5.3 Performance

In this subsection, we evaluate the performance overhead of ASLR-GUARD using the SPEC CPU2006 benchmarks. The results are shown in Table 20. Note that, for fairly comparison, we used the same basic compilation options (e.g., `-maccumulate-outgoing-args` and `-mno-push-args`) and only added "-aslrguard" and "-ffixed-r15" options in ASLR-GUARD build. The results are the average numbers over 10 executions. On average, ASLR-GUARD almost imposed no runtime (< 1%). In particular, 7 of them have even better performance than the original ones. gcc intensively calls the encryption routine of ASLR-GUARD, so its

performance overhead is about 10%. Compared with existing solutions for preventing code reuse attacks, ASLR-GUARD is more efficient. For examples, the original CFI imposes 21%([3]), bin-CFI imposes 8.5%([194]), CCFIR imposes 3.6%([193]), readactor imposes 6.4% ([48]), and CPI imposes 8.4%([98]) overhead. Compared with safe-stack of CPI [98], performance of AG-Stack is slightly better, which is even better than the original one. With this negligible runtime overhead, we believe ASLR-GUARD is a practical system to harden programs with information leaks. We also performed load-time overhead evaluation for ASLR-GUARD, which is shown in Table 20. The load-time overhead is as expected, as ASLR-GUARD performs remapping, relocation, and encryption in load-time. However the absolute time, i.e., 1 μs, is quite small. For space overhead, we measured the file size, which is 6% increased in ASLR-GUARD. The memory space overhead has two megabytes more for safe vault.

4.1.6 Discussion

Sophisticated attack defense. There are several sophisticated attacks proposed recently, e.g., "missing the point" [65] and COOP [153]. As the size of safe vault in ASLR-GUARD is only 2MB, there is no "always allocated" memory in ASLR-GUARD. Also offsets between sections remapped by ASLR-GUARD are randomized. Provided these two features, "missing the point" attack is not applicable to ASLR-GUARD. Regarding the COOP attack, as mentioned in §4.1.3.3, all virtual table pointers are encrypted as code locators, so attackers cannot know the locations of encrypted virtual function pointers to find vfgadgets, and thus ASLR-GUARD can eliminate the COOP attack.

Reusing encrypted code locators. An astute reader may point out that in theory, all the encrypted code locators that are stored in unsafe memory can leaked and reused to construct code reuse attacks. However, to reuse these code locators, attackers have to conquer the following challenges: (1) exploiting an arbitrary read vulnerability to leak the whole unsafe memory; (2) understanding the semantics of leaked memory, e.g., recovering

object boundaries, types and which functions the leaked encrypted code locators point to; and (3) preparing parameters for the target function. Step (1) is relatively easy, if the vulnerability can be repeatedly exploited without crashing the program. However, step (2) is non-trivial. Previous works that tries to reverse engineer the data structures in a memory snapshot either require an un-randomized anchor to bootstrap the traverse [29]; or require execution traces [108]. But neither requirement is satisfiable under our attack model. Step (3) is also challenging in x86-64 platform, where parameters are passed via registers. Although a recent research has demonstrate using forged C++ object to pass parameters, as discussed in §4.1.5.2, ASLR-GUARD can defeat such attack.

Despite raising the bar for attacks, we think ASLR-GUARD can be further improved in two directions: (1) reducing AIT. According to MCFI [136], binding code locator with `type` can significantly reduce the AIT to less 1% of original one. (2) code locator lifetime. We observe that many of the remaining code locators should only be dereferenced under a very specific context (e.g. pointers generated by `setjump()` should only be used in `longjump()`), and their lifetime should be limited and easy to determine.

Timing side-channel attacks. Timing attacks [65, 158] can infer the bytes at a relative address in unsafe memory or a guessed random address. ASLR-GUARD does not rely on the default OS ASLR, instead it generates new random base addresses and specifies them in `mmap`, so code sections are always randomized independently, which thus has 28-bits `effective entropy` [79]. As the size of safe fault and code sections are small (say 100 MB), rewriting the data pointer with a guessed random value only has a probability of $1/2^{14}$ to hit the mapped code or safe vault. The probability can be further reduced by improving the entropy of the generated random base. Also, since all code locators in unsafe memory are encrypted, such timing attacks can only obtain the encrypted code locators. More importantly, our encryption scheme has the same computation costs for different locators, so it is resistant against timing attacks.

Dynamic code generation. Besides static code, recent research also showed the feasibility

of launching code reuse attack targeting dynamically generated code [10]. Our current implementation of ASLR-GUARD cannot prevent such attacks. However, we believe our current design can be naturally extended to protect dynamically generated code. Specifically, since the code cache used to store the generated code is already randomized (as code sections), we only need to identify pointers that may point to the code cache. This can be done through a combination of our memory analysis tool and analyzing the code of the JIT engine.

4.1.7 Related work

Pointer integrity. The most related work to ASLR-GUARD is PointGuard [43], which also employed the idea of encrypting code pointers. However, it uses a single key to encrypt all code pointers with XORing, which is vulnerable to chosen-plaintext attack and forgery attack. Moreover, its type-based analysis cannot cover non-pointer code locators. AG-RandMap overcomes all these problems without sacrificing performance.

Code pointer integrity [98] employed type-based analysis to identify code pointers and unsafe stack objects, which misses non-pointer code locators. The 64-bit implementation of CPI also relies on randomization to protect its safe region. Because its safe region contains metadata of all recursively identified code pointers and safe stack, its size is so huge that there is an "always allocated" memory. In addition, the mapped sections in CPI have fixed offsets to each other. Due to these two issues, CPI is demonstrated to be bypassable by the "missing the point" attack [65]. On the contrary, the size of safe vault in ASLR-GUARD is much smaller (2^{21} vs 2^{42}) and section-level randomization is performed 4.1.3.2. And thus "missing the point" is mitigated by ASLR-GUARD.

Fine-grained randomization. Under the standard ASLR deployed in commodity OS, the leak of a single address allows attacker to derive addresses of all instructions. To address the shortcoming, many finer-grained randomization schemes have been proposed. [80, 142] work at instruction level, [92, 187] work at basic block level, and [12] works at page level.

Unfortunately, as demonstrated in [168], all these fine-grained randomization schemes are vulnerable to attacks that leverage a powerful information leak. Since ASLR-GUARD aims to overcome the weakness against information leak, it is orthogonal to these work and can be deployed together to protect them from information leak based attacks.

Dynamic randomization and re-randomization. JIT-ROP [168] circumvents all fine-grained code randomization techniques by continually disclosing the code pages, assuming the code memory layout is fixed at runtime. To break this assumption, Isomeron [54] and Dynamic Software Diversity [46] dynamically choose targets for indirect branches at runtime to reduce the probability for adversary to predict the correct runtime address of ROP gadgets. However, the security of such scheme depends on the number of gadgets required to perform a ROP attack. If a small number (say 2) of gadgets is enough to perform the attack (e.g., calling `mprotect()`), the attack still has a high chance to survive (e.g., it is 25% in Isomeron). OS-ASR [71] prevents memory disclosures by live re-randomization. However, the effectiveness and performance are dependent on the frequency of re-randomization. For example, JIT-ROP can finish the attack as fast as 2.3s. To prevent such attack, OS-ASR need provide a re-randomization with interval less than 2.3s. Based on its evaluation, an interval with 2s will impose about 20% runtime overhead. In ASLR-GUARD, we completely prevent attackers reading the code sections and code pointers, which provides a better security. Also ASLR-GUARD imposes a negligible overhead.

Execute-only memory. Another approach to defeat information leak attacks is to combine fine-grained randomization and execute-only memory [11, 48]. However, they all require modification to the operating system. As a comparison, ASLR-GUARD does not require any OS modification thus is more practical. Moreover, leaking the address of trampoline table in Readactor may decrease the security provided by it to the one of coarse-grained CFI, as all entries in the table may be reused. To mitigate this issue, Readactor performs *register allocation randomization* and *callee-saved register reordering*. However the semantic of the whole function are remained the same. So *return-into-libc* attack still works under Readactor.

ASLR-GUARD prevents all code reuse attacks, including return-into-libc.

Control flow integrity. Control flow integrity (CFI) [3] is considered as a strong defense against *all* control-flow hijacking attacks. Recent works mainly focused on making this approach more practical, eliminating the requirement of source code, reducing the performance overhead, providing modular support, or integrating into standard compiler toolchain [136, 178, 193, 194]. Unfortunately, many of these CFI implementations are bypassable [31, 55, 72]. Furthermore, CFI-like code reuse attack detection techniques (e.g. [38, 143]) have also been found to be bypassable [55]. Compared with these works, ASLR-GUARD's encryption scheme can provide similar or better effectiveness for reducing the possible code reuse targets §4.1.5.2, and has lower performance overhead §4.1.5.3.

Type-based CFIs [8, 86, 136, 178] reduce the average indirect targets (AIT) by checking indirect branches with type. In particular, MCFI [136] checks type and number of parameters, Forward-edge CFI [178] checks the number of parameters, Safedispatch [86] and vfGuard [8] check class type for virtual function calls. All these protections are orthogonal and inspiring to ASLR-GUARD. In the future, we will also bind type to indirect branches to further reduce AIT. Opaque CFI [127] relies on static analysis to identify targets for indirect branches, and insert checks to limit each branch in the range from the lowest address to the highest address of all targets. So the security is various based on the range of bounds. Another very related work is Crypto-CFI [123] which encrypts all code pointers and return addresses. As a result, it imposes a significant overhead (45%). On the contrary, ASLR-GUARD leverages efficient protection techniques for stack and encryption, thus has a much lower performance overhead (< 1%). HAFIX [9] achieves fine-grained backward-edge CFI in hardware with a reasonable performance overhead (2%). ASLR-GUARD not only protects return address but also function pointer without the need of hardware support, and its performance is even better.

4.2 BUDDY: *Detecting Memory Disclosures with Replicated Execution*

Modern systems widely employ randomization-based security mechanisms (e.g., ASLR) to prevent a variety of attacks such as code-reuse and privilege escalation attacks. These attacks generally require overwriting a code/data pointer with a malicious one. To achieve this goal, attackers have to know both the address of the target pointer and the address of malicious code pieces (i.e., the value of the malicious pointer), whereby both of them are randomized by ASLR. However, the randomized addresses are susceptible to memory disclosures. Attackers have exploited memory disclosures to bypass ASLR. For example, recently proposed advanced code reuse attacks [66, 106] all leak a randomized pointer to bypass ASLR and then compromise the fine-grained control-flow integrity techniques [136, 178]. In addition, memory disclosures also directly result in the loss of sensitive data such as private keys. For example, the HeartBleed [62] vulnerability allowed attackers to remotely read sensitive memory regions from 24-55% popular HTTPS sites. Even worse, according to vulnerability databases [57], memory disclosures are very common, and its number is increasing. As a result, memory disclosures are posing a rising threat to our modern systems.

The most common technique [20, 157, 167] for disclosing memory is exploiting memory errors such as buffer over-read. In response, many defense techniques [19, 49, 110, 120, 131] have been proposed to prevent memory disclosures. In particular, Readactor [49], ASLR-Guard [120], and TASR [19] transform or track code pointers to prevent them from being leaked. SeCage [110] protects private keys by storing them in an isolated memory region. Memory safety techniques (e.g., SoftBound [131], CETS [132], and AddressSanitizer [160]) provide a strong protection by preventing memory errors that can cause memory disclosures. Unfortunately, these defense techniques all suffer from several shortcomings. Specifically, code-pointer-only protections leave the *data pointers* and other sensitive data unprotected. Note that consequences from leaking data pointers can be equally (if not more) severe as recent attacks have shown how data pointers can be hijacked to launch attacks. For example, Stackdefiler [106] showed that fine-grained control-flow integrity can be bypassed

by leaking the stack address or modifying virtual function table pointers. Isolation-based protections [110] save the target data in an isolated memory, and only the pre-defined trusted code can access it. Such approaches suffer from two limitations: (1) finding all the trusted references to the target data in complex programs is hard and (2) the target data can still be leaked if the "trusted code" itself contains vulnerabilities. Memory safety techniques are effective in preventing memory errors caused by memory errors; however, they require re-compilation of all code, and their significant performance overhead (about 2x) impedes the adoption. In general, preventing memory disclosures caused by memory errors is very challenging. End-to-end preventions (e.g., code pointer protection) suffer from significant false negatives and low extensibility, while general solutions (e.g., memory safety techniques) suffer from high performance overhead and deployment issues.

In this work, we propose BUDDY, a runtime protection framework that effectively detects memory disclosures caused by memory errors. BUDDY can work on COTS binaries directly. The main idea behind BUDDY is that by seamlessly maintaining two running instances and diversifying only the target data, we can always detect any leak of such data because the leak will result in the two instances outputting different values. The two buddy instances are well synchronized in such a way that they act as a single instance from the perspective of external attackers. The only difference between the buddy instances is the diversified target data. To avoid false positives, we (1) synchronize these buddy instances by thoroughly virtualizing syscalls, virtual syscalls, and instructions that may return different values (i.e., non-deterministic sources) to the buddy instances, and (2) only synchronize buddy instances and detect divergence at I/O write: BUDDY reports leaks only when the disclosed data leaves OS. This way, the false positive issues caused by data structure padding [117] are eliminated. With this design, in principle, BUDDY can completely (i.e., no false negatives) and accurately (i.e., no false positive) prevent any leak without the need to identify or track the data in memory, which can be error-prone and expensive.

BUDDY is a general framework that mitigates memory disclosures caused by memory

errors through replicated execution. To realize such a framework, we first develop a formal model for BUDDY and define two properties that a BUDDY system should satisfy. To illustrate the effectiveness of BUDDY, we then develop multiple schemes on top of BUDDY to comprehensively detect memory disclosures caused by various memory errors. Specifically, we propose `partitioned address randomization` to detect memory disclosures caused by absolute address-based over-reads. The partitioned address randomization scheme ensures that the address spaces for buddy instances are non-overlapping and randomization enabled. Any absolute address-based over-read will already result in one instance crashing, thus triggering detectable divergences. Further, to detect memory disclosures caused by relative address-based over-reads, we develop the *random padding* scheme that introduces paddings with random values among both stack frames and heap objects. Whenever such an over-read occurs, different data values will be obtained; therefore, leaking them will also trigger the divergence detection of BUDDY. To detect memory disclosures caused by temporal memory errors (e.g., use-after-free), we borrow the randomization scheme from Diehard [16] that allocates objects in a random address, so that an uninitialized value or a defined value after free will likely differ in buddy instances; leaking the value will trigger the divergence detection of BUDDY. Applying all these schemes on top of BUDDY, we are able to comprehensively detect memory disclosures caused by various of memory errors. For example, BUDDY can effectively detect the emerging address leaks and the Heartbleed attack.

BUDDY-based memory disclosure detection has three important advantages. First, the error-prone and expensive data tracking are completely avoided. That is, whenever the diversified data is leaked through I/O by a remote attacker, we can detect it. Second, in principle, BUDDY-based detection can detect leaks without false negatives. That is, as long as the returned data from the buddy processes contains any diversified data, they must be different, otherwise, the leak is not meaningful for the attacker to infer the data. Third, BUDDY can directly work on binaries without the need to recompile the source code. Note

that, although BUDDY double-executes the user-space code (kernel code and drivers are executed only once), its performance overhead is amortized with multi-core CPU that is commonly deployed in modern devices. As shown in §4.2.6, BUDDY only imposes a small performance overhead.

Although BUDDY leverages the n-version approach [35], *its goal is not to build an enhanced n-version system, instead it aims to use the n-version approach to achieve a new security goal—detecting memory disclosures caused by memory errors*. To this end, we have proposed multiple new mechanisms such as single syscall synchronization (Table 4.2.4.2), partitioned address randomization (§4.2.4.1), and random padding (§4.2.4.1). We differentiate BUDDY from many previous n-version systems [16, 45, 69, 81, 94, 100, 124, 183] that were mainly designed to detect intrusions or errors. First, since intrusion or error may happen at any point during execution, these n-version systems (e.g., n-variant systems [45]) have to intercept and synchronize *all* syscalls. A fundamental problem with such a design is that synchronizing every syscall is expensive because of the frequent waiting and notifying between variants. More seriously, synchronizing every syscall may cause false positives. For example, divergent data caused by uninitialized memory (it is common due to data structure padding [117]) should not be treated as a leak if it does not leave the OS through I/O write. ReMon [183] improves the efficiency by eliminating the synchronizations of non-sensitive syscalls; however, its `ptrace`-based syscall interception requires extra context switches, which can cause considerable runtime overhead. BUDDY only synchronizes and detects memory disclosures with a single synchronization point—I/O write (i.e., file write and socket write) and thus minimizes performance overhead and false positives. Second, traditional n-version systems typically detect intrusion or error by monitoring divergence in the syscall sequence, which is not applicable to information leak detection because information leak does not necessarily change the control flow. BUDDY strictly synchronizes at only I/O write and detects divergence in the outgoing contents in a fine-grained manner.

We implemented the BUDDY framework on the Linux 64-bit platform. BUDDY is a

practical system that does not require the availability of source code of a target program, modification of compilation toolchain, or recompilation of OS kernels. Instead, it directly supports any COTS program. BUDDY mainly consists of (1) a kernel space coordinator that hooks the syscall table to intercept and synchronize syscalls, (2) a user space coordinator that synchronizes virtual syscalls and instructions in user space, and (3) a controller that prepares both coordinators, starts and monitors the buddy instances. With the BUDDY framework, we further implemented two diversification schemes for detecting memory disclosures, the partitioned address randomization scheme and the random padding scheme.

We evaluated the performance of the BUDDY framework with the partitioned address randomization scheme and the random padding scheme using the SPEC CPU2006 benchmarks and multiple server programs, including Apache web server, Nginx web server, OpenSSL, PHP, Lighttpd, and Orzhttpd. For effectiveness evaluation, we evaluated BUDDY using a set of real address leak attacks and the HeartBleed attack. The evaluation results show that BUDDY can effectively prevent all tested data leaks. Since BUDDY is a 2-variant system that runs user-space code in parallel, for a single-core processor, BUDDY may incur a performance overhead of more than 100%. However, with a multi-core processor, BUDDY's performance overhead is small: 2.3% on the SPEC benchmark and around 4% on web server programs. Moreover, the partitioned address randomization scheme introduces a non-measurable overhead, and the random padding scheme introduces an additional 2.8% overhead. Even in the scenario where the CPU is highly loaded already (e.g., 99% CPU usage), BUDDY introduces a performance overhead of only 8.3% on the SPEC benchmarks.

In summary, this work makes the following contributions:

- We proposed BUDDY, a framework for detect memory disclosures caused by memory errors. BUDDY avoids the expensive and error-prone date tracking process.
- We proposed the adaptive ring buffer based virtualization mechanism (with a single synchronization point) to boost the performance of the buddy execution and minimize its false positives.

- Based on the BUDDY framework, we designed the partitioned address randomization and the random padding scheme to detect memory disclosures caused by spatial memory errors in general.

4.2.1 Challenges in Detecting Memory Disclosures

A common way for attackers to disclose memory is to exploit memory-error vulnerabilities. Such vulnerabilities can be spatial such as out-of-bound read or temporal such as uninitialized-data read and use-after-free. According to CVE details [57], the number of memory disclosure-related memory errors is still increasing in general. Researchers have attempted to mitigate memory disclosures with various approaches, which can be classified into four categories: data-flow tracking, memory isolation, data encryption, and memory error elimination. We now discuss these approaches and their limitations.

Data flow tracking. Data flow tracking aims to detect illegal data usages (e.g., data leak). Static data flow tracking performs a reachability analysis from the data sources to the sinks. Since such analyses are guided by the pre-built data dependency graph, they cannot catch leaks that are out of the graph (e.g., flow diverted by memory errors). Dynamic data flow tracking [155] is accurate in detecting leaks; however, its orders of magnitude of the runtime performance overhead is unacceptable to be used in production.

Memory isolation. Memory isolation approaches [110] save the target data in an isolated memory region, and only the pre-defined trusted code can access it; therefore, external attackers cannot touch the isolated memory. Two fundamental limitations with these approaches are: (1) it is difficult to find all the trusted references to the target data in complex programs; and (2) the target data in the isolated memory still can be leaked if the "trusted code" itself contains vulnerabilities.

Data encryption. Data encryptions are extensively used to protect the target data, which aim to render the disclosure of the memory useless in inferring the content of the target data. Effectiveness of such approaches relies on the confidentiality of the encryption key. Further,

data encryption is expensive and thus not suitable for protecting data (e.g., data pointer) that is prevalently scattered over the memory. More importantly, encryption-based protections may suffer from replay attacks. Attackers can replace an encoded pointer with another to hijack the control flow.

Memory-error elimination. Memory safety techniques [131] can eliminate memory errors, so memory disclosures caused by them are naturally prevented. However, such techniques require modification of toolchain and runtime, and usually impose a significant performance overhead (e.g., SoftBound [131] imposes a 70% runtime overhead).

Conclusion: While preventing memory disclosures is a pressing security goal, for practical deployment, it is very challenging to achieve *effectiveness, completeness, and efficiency*.

4.2.2 Formal Model

We aim to detect attacks that exploit memory errors to disclose memory data. As discussed in §4.2.1, existing approaches are not adequate in detecting memory disclosures, hence we need a new solution that is general, effective, and efficient. To this end, we propose BUDDY, a new framework for detecting memory disclosures. At high level, BUDDY maintains two identical instances (i.e., buddy instances), P and P', for an original process P_o, and diversifies the target data in the two buddy instances. BUDDY detects disclosures by monitoring divergences when the diversified data values are disclosed through some points such as I/O write. This way, the expensive and error-prone data-flow tracking is avoided. In this section, we present the formal model of BUDDY.

We first define `detecting points`: [D_0, D_1, D_2, ...]. In BUDDY, detecting points trigger detection for memory disclosures. Examples of detecting points include I/O write system calls. We then view an execution of a process as a potentially infinite sequence of states: [S_0, S_1, S_2, ...]. In BUDDY, at detecting point i, we have a pair of states for the two buddy instances: <S_i, S'_i>. Therefore, for detecting points from 0 to i, the execution of the two buddy instances is represented as: [<S_0, S'_0>, <S_1, S'_1>, <S_2, S'_2>, ...]. Note that the

sequence of states at detecting points is a subset of the whole state sequence of a process.

States of buddy instances in each state pair are semantically equivalent. However, BUDDY diversifies `non-semantic attributes` such as memory layout (we assume the semantics of a process are independent from memory layout) in the two buddy instances. Besides semantic equivalence, the BUDDY system also maintains the mapping from the states (i.e., <S_i, S'_i>) of buddy instances P and P' to the state (i.e., $S_{o,i}$) of original process P_o through mapping function $Map()$, at detecting point i. The original process has a transition function T_o, and the two buddy instances have transition functions T and T', respectively. All these transition functions take an input and a state at a detecting point, and then produces the state at next detecting point. Note that we will ensure that the state at the last detecting point represents the last state of a process execution for an input.

The equivalence property under normal execution. Under normal execution (i.e., no memory disclosure), we have the following induction:

$$Map(S_0) = Map(S'_0) = S_{o,0}. \tag{1}$$

$$\begin{aligned} &\forall 0 \leq i \leq N, \forall I \in Normal\ inputs: \\ &Map(T(S_i, I)) = Map(T'(S'_i, I)) = T_o(S_{o,i}, I). \end{aligned} \tag{2}$$

The above equations shows the following property, namely `equivalence property`, under normal execution when we assume the two buddy instances are synchronized at each detecting point.

- Equation 1 shows that both two buddy instances are in the equivalent initial states that can be mapped to the state of the original process.
- Equation 2 shows that, given an input, the transition functions in both two buddy instances preserve mapped equivalence at each detecting point.
- Since the last detecting point is the end of the execution, Equation 2 also shows that

both instances produce the same outputs as the original process, ensuring semantic equivalence.

The divergence property under memory disclosures. We assume that the initial states of buddy instances are not under attack, so Equation 1 is always satisfied. However, whenever an attacker discloses some memory data, we need to ensure a property, namely `divergence property`, so that Equation 2 is not satisfied. Formula 3 shows that if some memory disclosures occur during transition right after detection point `i`, the states at detection point `i + 1`, S_{i+1} and S'_{i+1}, must be different. With the divergence property, we detect memory disclosures in this way: Whenever the states of the two buddy instances are not equivalent, we know there is a memory disclosure attack.

$$\begin{aligned} &\forall 0 \leq i \leq N, \forall I \in Inputs : \\ &T(S_i, I)) \; or \; T'(S'_i, I)) \in Memory \; disclosures \Longrightarrow Map(S_{i+1}) \neq Map(S'_{i+1}). \end{aligned} \tag{3}$$

Threat model. We have modeled the BUDDY system, including its properties and how we detect memory disclosures. To make the model as practical as possible, we make the following main assumptions:

- Attackers have to disclose memory through pre-defined detecting points such as I/O write system calls.
- The target program will not intentionally use unspecified memory such as uninitialized memory as part of its semantics.
- We have the list of non-determinism sources so that we can virtualize them and ensure the divergence of Equation 3 is caused by memory disclosures.

Moreover, we focus on preventing remote attacks that disclose memory by exploiting memory errors (not logic errors). After gathering the memory data, the attacker may further launch subsequent attacks, such as code reuse attack. We assume the hardware and the OS kernel as our trusted computing base (TCB), so attacks that target the hardware (e.g., cold

boot attack [75]) and the kernel are excluded. Since there are many ways to acquire data of a program *locally* (e.g., by directly reading `/proc/pid/mem` and cache side-channels), we *do not* consider attacks where attackers already have the ability to execute arbitrary code. BUDDY currently does not support web browsers; however, the concept of BUDDY is general, and we discuss how to support script environments with BUDDY in §4.2.7. Side-channel-based leaks are still possible because we are not able to cover all detecting points of side channels. The target program may contain one or more vulnerabilities that can be exploited to disclose memory. The program's source code and binary may be available to attackers, and they can perform both static and dynamic analyses. We do not limit the types of vulnerabilities (except side-channels); they can be buffer over-read, format string, reading uninitialized memory, etc. For defense mechanisms, we assume that the target program is PIE-enabled (position-independent-executable) and the underlying operating system enables both ASLR and DEP (i.e., no code injection attack). Finally, we assume that the processor is multi-core.

4.2.3 The BUDDY Approach

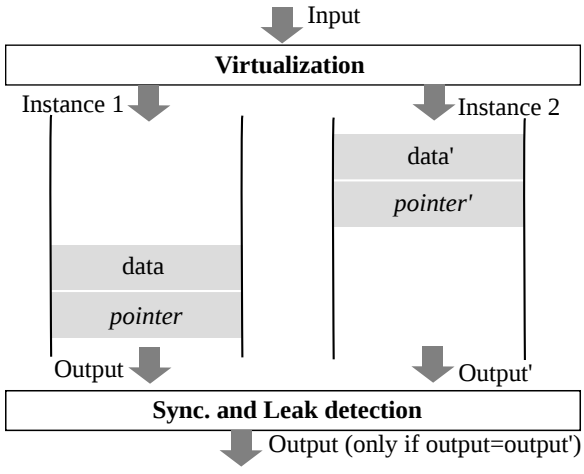

Figure 30: Overview of the BUDDY approach. BUDDY virtualizes all virtualizing points and strictly synchronizes at I/O write to detect any divergence in outgoing data. For example, If $pointer$ and $pointer'$ are disclosed, BUDDY is able to detect it at socket write because they are not equal.

Figure 30 illustrates how BUDDY works: whenever the target program is about to run,

BUDDY launches two identical instances (processes). The diversification schemes ensure that reading the target data in these buddy instances through memory errors will get different values. Now let us consider a concrete scenario where the attacker sends a malicious payload to disclose the target data. BUDDY duplicates and sends the payload (input) to both instances. To make sure both instances exhibit the same semantics during payload processing, all syscalls, virtual syscalls, and instructions that may return different results (i.e., non-determinisms) are virtualized to always return the same value to both instances. After processing the attacker's payload, both instances generate the corresponding outputs and write them to the socket. At this moment, by hooking I/O write syscalls, BUDDY captures the data to be sent out and compares their contents. If both outputs are the same, BUDDY continues the write operation (but only performs *once*); otherwise, a divergence is detected, and BUDDY indicates that the target data is being disclosed.

Based on the above scenario, to achieve memory disclosure detection, BUDDY needs to include the following three parts: (1) non-interference diversification that diversifies the target data but will not break program semantics, (2) virtualization of buddy instances, which properly intercepts non-determinisms to ensure that they always return the same value, and (3) disclosure detection that captures divergences at I/O write.

4.2.4 Design

In this section, we present the design of the three key components of BUDDY: non-interference diversification, coordination of buddy instances, and memory disclosure detection.

4.2.4.1 Non-Interference Diversification

BUDDY detects memory disclosures by catching divergences in outgoing data, which requires that the target data is diversified. Diversification should be performed in a non-interference manner to satisfy the `equivalence property`, i.e., during normal execution, the diversification should not break the semantics of the target program or cause any difference

Table 21: Non-interference diversification schemes.

Category	**Spatial (Buffer over-read)**			**Temporal**	
Type	Absolute addr	Relative addr		Uninitialized read	Use-after-free
		Continuous	Offset-based		
Diversification	Partitioned ASLR	Random padding	Random padding	Diehard [16]	Diehard [16]
Equivalence property	✓	✓	✓	✓	✓
Divergence property	✓	Probabilistic	Probabilistic	Probabilistic	Probabilistic

in outputs. Further, the diversification scheme should also satisfy the `divergence property`, i.e., any memory disclosures can be detected. In this work, we design two diversification schemes. The `partitioned address randomization` scheme detects any over-reads that are based on an absolute address. The `random padding` scheme detects continuous over-reads that are based on a relative address. When an over-read is offset-based (i.e., skipping some data), the `random padding` scheme provides probabilistic detection. To detect memory disclosures caused by temporal memory errors (e.g., use-after-free), we borrow the diversification scheme from Diehard [16], which allocates objects in random addresses. Note that, BUDDY is a general framework; more diversifications such as fine-grained ASLR [12, 92] can be applied to BUDDY to achieve other security properties.

Partitioned Address Randomization. The scheme we propose to detect absolute address-based over-reads is `partitioned address randomization`. We first equally partition the available address space into two non-overlapping sub-spaces. The two buddy instances are then loaded into the sub-spaces, respectively. We retrofit the loader (`ld.so`) to ensure that memory layout randomization is also enforced separately in each sub-space. Compared to the default ASLR, our randomization has 1-bit less entropy because of the partitioning.

Equivalence property. The position-independent code (PIC) or position-independent executable (PIE) technique in modern compilers already allows us to map the process memory into a random address, so our partition address randomization is non-interference.

That is, it satisfies the `equivalence property`.

Divergence property. Since the sub-spaces are non-overlapping, we ensure that any absolute address-based over-read will result in one instance crashing, and we can detect it. Therefore, such a scheme also satisfies the `divergence property`.

Random Padding Scheme. To detect memory disclosures caused by relative address-based over-reads, we propose the random padding scheme. The design of the random padding scheme is based on our observation that an relative address-based over-read will get different values if we append different paddings to the target data in the buddy instances so that BUDDY can detect divergences when the overread data are disclosed through I/O write. Figure 31 shows the design of the scheme: in one instance, the padding consists of an 8-byte random value and a 24-byte undefined memory (i.e., a placeholder); in the other, the padding consists of an 8-byte random value and an 8-byte undefined memory. The size of the padding is set to either 32 or 16, in order to conform to the data alignment requirement. The sizes of the padding in two buddy instances are set to be different so that an offset-based over-read (i.e., discontinuous over-read) will also get different values. The random value in the padding is set to the value of register `RIP`, for two reasons: (1) values of `RIP` are guaranteed to be different because the base addresses are different, so an over-read of the paddings will always get different values; (2) compared to using real random value, using `RIP` is much more efficient.

For stack, the padding is inserted between stack frames (right after return address). x86_-64 calling convention passes the first 6 parameters through registers, but the remaining ones are still passed through stack. As such, we also need to update all the parameter-accessing instructions in the function by patching the offsets. For heap, the padding is inserted between the metadata (i.e., head of the object) and the actual object memory. The padding can be enforced in a finer-grained manner (e.g., field level); however, it will incur more performance overhead. We currently choose object-level and stack-frame-level padding for a better performance.

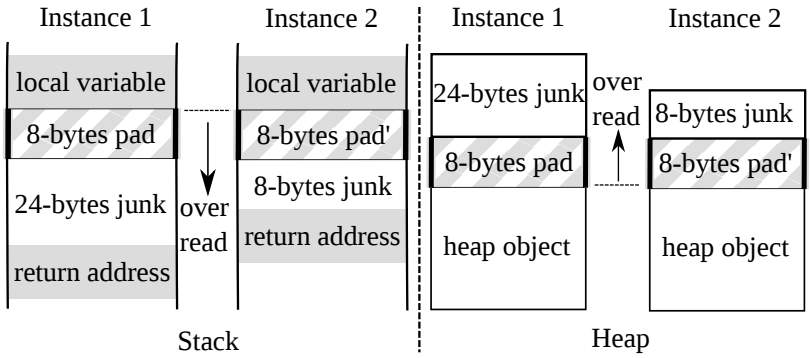

Figure 31: The random padding scheme for stack and heap objects. Random value is inserted between any two stack frames or heap objects. The size difference between placeholders ensures different offsets.

Equivalence property. Similar to the partitioned address randomization scheme, our random padding scheme does not change the original semantics of a program. Instead, it only changes the memory layout of objects in stack and heap, and updates the offsets in instructions to ensure proper accesses. Therefore, we establish the equivalence property in the random padding scheme.

Divergence property. Continuous over-reads will always read the inserted padding bytes. Since we guarantee that the padding bytes in the two buddy instances have different values, such over-reads will always get different values. Therefore, whether the divergence property is satisfied depends on the `computation` between the over-read and the next detecting point. If the computation is a deterministic, one-to-one function (such as in the Heartbleed attack [62]), we can guarantee that the outputs from the two buddy instances at the next detecting point must be divergent, and thus the divergence property is satisfied. However, if the computation is non-deterministic or n-to-one function, it is possible that the outputs are the same at the next detecting point. In this case, the divergence property is probabilistic. As for offset-based over-reads, we cannot guarantee the divergence property because such over-reads may skip the inserted padding bytes and junk bytes. Whether the read data differs depends on the values of the target data. If the target data is random, the read data has a probability of $2^N - 1$ out of 2^N to differ, where N is the number of read bits. However, if the target data is a sequence of a repeating bytes, attackers may read the same

value at the diversified offsets, bypassing the scheme. Such a case is not common for secret data such as encryption keys and randomized address of ASLR.

4.2.4.2 *Coordination of* BUDDY *Instances*

The goals of coordination are to ensure that (1) the buddy instances always receive the same inputs and (2) any operation that has an external impact to the program is executed only once. In this way, we ensure the BUDDY system behaves as a single entity to attackers, and whenever an divergence is detected at output, we can conclude with confidence that the target data is being disclosed.

Virtualizing Points. We need to virtualize the execution of BUDDY for two reasons: (1) the two buddy instances are maintained to be identical except the intentionally enforced diversity (e.g., randomized memory layout), so all non-deterministic operations should be virtualized to return same values; (2) BUDDY replicates the execution for only user space, so non-user-space operations such as kernel space operations should also be virtualized to return same values to the two buddy instances. In this work, we use the term *virtualizing point* to refer to a syscall, a virtual syscall, or an instruction that may return non-deterministic values (e.g., `RDRAND` instruction) or different values (e.g., `open` syscall) to the buddy instances, or have observable external effect (e.g., `write` syscall). Using this definition, to make sure that both buddy instances exhibit the same semantics and act as a single application to external entities, we need to make all the virtualizing points return the same values to the buddy instances, except memory-layout-related ones such as `mmap`. Please note that identifying all virtualizing points is not a new problem, which is necessary for all n-version systems [16, 27, 45, 81]. Similar to these previous systems, we also conservatively include most syscalls, all virtual syscalls, and non-deterministic instructions as the virtualizing points. The summary of virtualizing points is shown in Table 22.

Completeness of virtualizing points. Although both traditional n-version systems and BUDDY have conservatively selected the virtualizing points, we still cannot claim they are

Table 22: Included virtualizing points. Besides memory space related syscalls, they are conform to previous N-version systems [16, 27, 45, 81].

Category	Virtualizing points
Syscall	All syscalls except memory space related ones (e.g., `mmap`)
Virtual syscall	All virtual syscalls
Instruction	`RDTSC` and `RDRAND`

always complete for all programs because new non-determinisms (e.g., new instructions or new syscalls) may be added to the system over time, and some non-deterministic inputs or output channels may be unique to a system or a program (e.g., special hardware feature). If a new non-determinism is introduced, we need to include it as a virtualizing point. Please note that the completeness of current virtualizing points has been validated by extensive experiments over many real server programs. As shown in §4.2.6, BUDDY can reliably run these programs without triggering any real false positive. Therefore, we believe our current set of virtualizing points is arguably complete for practical deployment.

Intercepting Virtualizing Points. To coordinate a virtualizing point, we need to intercept it first so as to insert the virtualization logic. There are two requirements for intercepting virtualizing points. First, the interception must be reliable so that the attacker cannot bypass it. Second, since many operations may be frequently executed (e.g., `getpid` and I/O operations), the interception must be efficient. To satisfy both requirements, our interception employs different approaches to intercept syscalls, virtual syscalls, and instructions.

Syscall. There are many ways to intercept syscalls. A common way is to use `ptrace` [45], as it is available in most *nix systems. However, because `ptrace` introduces extra context switches for each intercepted syscall, this approach is not efficient enough. Therefore, instead of using `ptrace`, we chose to temporarily patch the syscall table using a kernel module. In this way, extra context switches can be avoided to improve performance. Note that patching the syscall table will also affect other programs, as such, we add a PID-based filtering scheme to only intercept the buddy instances. Since PID is obtained in the kernel

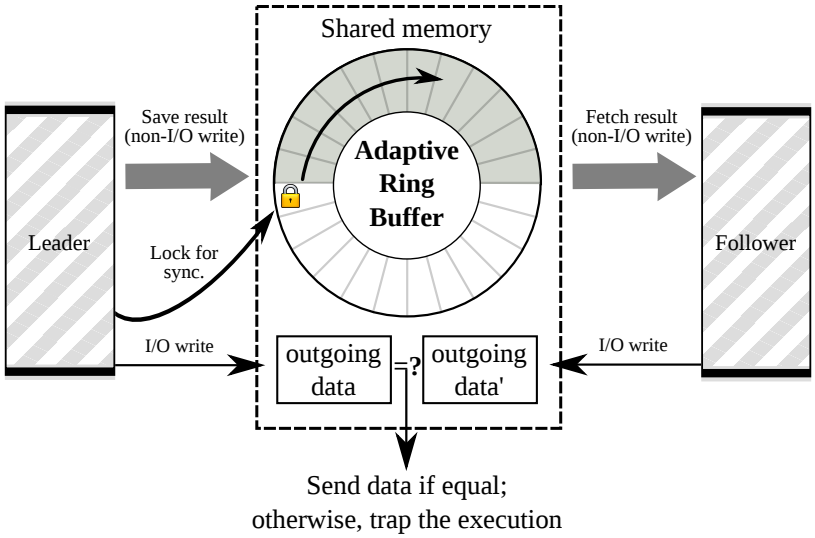

Figure 32: The adaptive ring buffer-based virtualization. Normal virtualizing points are virtualized in a ring buffer manner. However, upon I/O write, BUDDY locks the ring buffer and strictly synchronizes to detect memory disclosures.

module, the attacker cannot modify it.

Virtual syscall. To accelerate the execution of certain "read-only"syscalls like `gettimeofday`, kernel developers export a small set of functions to the virtual dynamic shared object (vDSO). Since virtual syscalls do not actually go through the kernel, the syscall table patching based interception is not applicable. For these syscalls (only four in x86_64 systems), we chose to patch the `GOTPLT` table that contains the entries to these virtual syscalls.

Instruction. In general, binary rewriting is necessary for intercepting particular instructions. But rewriting an instruction with another one with more bytes (e.g., `jmp`) may change the code layout. To reliably intercept instructions, we replace them with an one-byte interrupt instruction (e.g., `INT3`), which allows us to perform virtualization by handling the interrupt. At the same time, because the intercepted instructions like `RDTSC` and `RDRAND` are typically not executed frequently, our approach does not generate much more overhead than `jmp`-based interception.

The Adaptive Ring Buffer Virtualization. Traditional n-version systems perform coordination in a fully synchronized manner (i.e., lockstep), which would introduce significant overhead. For example, for Apache web server, because of frequent CPU swap, the fully

synchronized approach can impose a performance overhead of over 50%. To solve this problem, we take an approach similar to [81, 179]. Specifically, one of the buddy instances is assigned as the *leader* instance, and the other is assigned as the *follower* instance. The leader instance performs the actual operation and stores the results in a shared memory, while the follower just retrieves the results without actually performing the operation. In this way, from the perspective of the buddy instances themselves, they are independent instances; but from the perspective of an external entity (e.g., the attacker), they are a single instance. At the same time, because the leader does not need to wait for the follower, the performance overhead is much lower. However, in order to prevent memory disclosures, it is still necessary to synchronize at every I/O write operation and check for divergences in the outgoing data.

To implement our coordination scheme, we designed the *adaptive ring buffer*-based virtualization mechanism (Figure 32). The ring buffer has a configurable but fixed amount of slots shared by both the leader and follower. Each slot consists of an arguments holder and a result holder. For virtualizing points other than I/O write, the leader performs the operation and puts the result in the result holder of a ring buffer slot; and when the follower reaches the same execution point, it picks up the result without actually performing the operation. Please note that because both buddy instances are exactly the same, any divergence in the syscall sequence will also be a sign of attack and will result in termination. For I/O writes, the leader puts the content in the arguments holder and waits for the follower; and when the follower reaches the same execution points, it compares its own output arguments with the leader's and informs the leader of the result. If the outgoing data is the same, then the leader performs the operation and puts the result in the ring buffer; otherwise, the operations is aborted. It is also worth noting that because syscalls are handled in the kernel space, BUDDY utilizes two ring buffers, one for user space coordination (i.e., virtual syscalls and instructions) and the other for syscall coordination. Obviously, attacker cannot modify the ring buffer in kernel space. For the ring buffer in user space, because it is mapped into

randomized (and different) addresses in the buddy instances, and memory addresses are protected by BUDDY, attackers cannot modify the data in shared memory either.

Multi-processing/threading support. Thread scheduling is another important source of non-determinism for multi-threaded programs. To completely resolve this issue, synchronizing all shared memory accesses is necessary to ensure that the leader and follower have consistent views on shared memory whenever they issue a syscall. In other words, the follower should follow exactly the same order of shared data accesses as the leader. However, such a total order synchronization can hardly be achieved without a high performance overhead or special hardware support, as evidenced in the deterministic multi-threading (DMT) domain [15, 140]. Neither is it necessary for many of the multi-threaded programs.

In BUDDY, we implemented a lightweight (but potentially incomplete) support for multi-threaded programs. At a high level, BUDDY assigns each pair of leader-follower processes/threads of the buddy instances to the same *execution group*; and each execution group has its own ring buffer. Forking/cloning is monitored to maintain execution groups: when the leader instance forks a process/thread, the child process/thread automatically becomes the leader process of the new execution group; and the forked process of the follower instance automatically becomes the follower process in the new execution group. Note that if the follower does not fork, it is a divergence in syscall sequence and will be detected by BUDDY.

In fact, for daemon-like programs (e.g., a majority of server programs such as Apache, Nginx, sshd), BUDDY's lightweight solution is sufficient to eliminate the syscall sequence deviations caused by non-deterministic schedulers because, for those programs, each thread-/process is highly independent of others and hardly or never access shared memory. We have also verified that BUDDY does not raise false alerts when synchronizing the aforementioned daemon server programs (§4.2.6). Developing sophisticated DMT solutions is out of scope for this work. In fact, BUDDY can leverage the latest development in DMT domain by directly plugging-in many existing DMT solutions, such as [15, 52, 109, 115].

4.2.4.3 *Memory Disclosure Detection*

Once all the virtualizing points are virtualized, BUDDY strictly synchronizes I/O write (e.g., local file write and socket write) and performs a quick comparison on the outgoing data to accurately detect memory disclosures. In particular, for arguments of non-pointer type (e.g., size of the buffer), we compare their values; and for pointers, because their values have been randomized by ASLR, we compares the content of the buffers they point to. Note that if one of the buddy instances crashes, or the syscall sequence is different, BUDDY also treats them as divergences.

Although we assume the attack is remote, it is still necessary to check local file write operations. The reason is that some vulnerability may allow attackers to use local "storage" (e.g., files) as "trampoline" to disclose the data indirectly. For example, an attacker can cause the leader instance to write the diversified data into a local file first; then read it from file and send it out through network. If there is no check for divergence is performed, we will miss such attack.

4.2.5 Implementation

We implemented a prototype of the BUDDY framework on the 64-bit Linux, which consists of a kernel space coordinator, a user space coordinator, and an instance controller. We will further the random padding scheme.

4.2.5.1 *Diversifiers*

Partitioned address randomizer. The partitioned address randomizer has two tasks: partitioning the address space into two sub-spaces and randomizing memory layout of instances in their own sub-spaces. We implemented the randomizer by customizing the dynamic linker (i.e., ld.so). The only modification we made to the dynamic linker is instrumenting the call to the `mmap` syscall. We modify the `mmap` call to ensure that it always maps the memory to a random address in the partitioned sub-spaces. In default 64-bit

Linux systems, ASLR has 28-bit randomness entropy; the randomized base address is of the form `0x7f???????000`, where `?` bits are randomized. In our partitioned address randomizer, we partition the address space into two ranges: <`0x7f0000000000, 0x7f7ffffff000`> and <`0x7f7ffffff000, 0x7fffffff000`>, and randomize the memory layout of buddy instances in the two ranges by specifying a legitimate and random base address to the `mmap` system call.

Random padding diversifier. As shown in Table 4.2.4.1, we insert random paddings among stack frames and heap objects. We insert random paddings among stack frames by modifying the GNU assembler. Such an implementation can support COTS binaries as long as the disassembly is complete and reassembleable [184]. Specifically, we first identify function prologues with the `.cfi_startproc` label, and return instructions. At function prologue, we create the padding: subtracting `RSP` and setting random value with the value of register `RIP`, which is obtained with the `LEA` instruction. Before return instructions, we restore the stack. The `repz ret` instruction (equivalent to `nop; ret;`) is used to improve the AMD processor performance; we also treat it as a return. If a function contains a tail call, we do not enforce padding for it. Parameter-accessing instructions are identified by pattern "`offset(%rbp)`." We update `offset` by increasing its value by either 32 or 16, depending on which instance it is in. For heap object, we use `LD_PRELOAD` to intercept heap allocation functions. After that, we enlarge the size of the object by 32 or 16, and insert the random value after the metadata of the object.

4.2.5.2 Kernel Space Coordinator

The kernel space coordinator is implemented as a Linux kernel module. It is responsible for virtualizing syscalls, synchronizing I/O writes, and performing leak detection. Syscall interception is done through standard syscall table patching, i.e., replacing the original syscall handlers with our callbacks.

Syscall filtering. Since BUDDY patches the syscall table at kernel space, syscalls from all

processes will be redirected to our callbacks. To quickly differentiate the BUDDY processes from other processes, we implemented a PID-based process filtering scheme. The kernel space coordinator holds a hashtable with PID as keys and variant control bock (`vcb`) as values. A `vcb` holds critical information about the BUDDY processes, such as the address of the ring buffer, whether the process is leader or follower, etc. Upon program launch, the instance controller will first inform the kernel space coordinator of the PIDs of the buddy processes. Upon task fork, the newly created task is automatically added to the hashtable; and upon task exit, the corresponding hashtable entry is removed. By looking up the hashtable, we could reliably distinguish whether the entered process is a BUDDY process or not.

4.2.5.3 User Space Coordinator

The user space coordinator is implemented as a shared library, which is preloaded into both buddy instances upon program start-up. The user space coordinator is responsible for intercepting and virtualizing virtual syscalls and instructions.

Intercepting virtual syscalls and instructions. Because virtual syscalls are essentially `libc` function calls, we can hook them by replacing the corresponding entries in the GOTPLT table callbacks to ours virtualization function. To do so, we use `LD_PRELOAD` to prioritize our library so that old virtual syscall functions are overridden with ours. Beside virtual syscalls, we found another special syscall that may not invoke the kernel—the glibc wrapper function `getpid()`. Because the result of this syscall will not change, glibc caches the result in user space to improve the performance. In other words, after the first syscall of `getpid`, it becomes a virtual syscall. Therefore, we also hook `getpid()` by patching the GOTPLT table.

To intercept non-deterministic instructions (e.g., `RDRAND`), we patch the first byte of the instruction with `INT 3`. At runtime, when this interrupt is triggered, we virtualize it through the standard signal handling mechanism. For `RDTSC`, binary-rewriting is not

even necessary: we can trap for `RDTSC` by making timestamp counter non-readable (i.e., `prctl(PR_SET_TSC, PR_TSC_SIGSEGV, 0, 0, 0)`). Whenever the timestamp counter is read, a SIGSEGV will be triggered, allowing for our virtualization.

Ring buffer for synchronization. The ring buffers in the user space must be explicitly created in the shared memory and mapped (using `mmap`) into both instances. The logic of creating and mapping this shared memory is also implemented in the shared library.

4.2.5.4 Instance Controller

The instance controller program is responsible for starting, monitoring, terminating, and restarting the buddy instances. It first informs the kernel space coordinator of the incoming buddy processes and then sets the `LD_PRELOAD` environment variable to the shared library. It then forks two children processes and uses `execve` to launch the target program in each forked process. After that, the instance controller pauses and waits for status change of the buddy instances. In case any of the buddy processes is killed, the instance controller is waken up, prints alerts and executes corrective actions (e.g. restart the system) according to policy specified by users.

4.2.6 Evaluation

BUDDY is designed to be a general framework that can detect memory disclosures caused by memory errors. In principle, the effectiveness of BUDDY is guaranteed by the fact that the disclosure of diversified data will cause a divergence at some points, and BUDDY detects the memory disclosure by catching the divergence. Efficiency is also a faithful goal for BUDDY to be practical. We extensively evaluated the robustness, security, and performance of BUDDY.

- **Robustness.** Evaluating BUDDY on various programs to empirically show that it incurs no false positives and does not break the semantics of original programs.
- **Security.** Analyzing the security of BUDDY and validating it with multiple real exploits.

- **Performance.** Evaluating BUDDY's performance with two sets of benchmarks: the SPEC CPU2006 Benchmarks (CPU intensive) and ApacheBench (I/O intensive).

Experiment setup. All experiments were conducted on an eight-core server machine with a 3.60 GHz Intel Xeon E5-1620 CPU and 64 GB RAM running 64-bit Ubuntu 14.04 LTS with 3.13.0-63-generic Linux kernel. For evaluation on web servers, we dedicate a remote machine as the client and use it to simulate client requests as well as measure the request roundtrip time and data transfer rate. Server and client machines are connected in a university-level LAN.

4.2.6.1 Robustness

There are two goals of robustness evaluation: (1) to empirically validate the completeness of the set of virtualizing points in practice; (2) to validate that BUDDY do not cause any error. In particular, we ran BUDDY on the SPEC CPU2006 benchmarks and popular server programs, including Apache web server (configured to include PHP module and use OpenSSL), Nginx web server, Lighttpd, and Orzhttpd. For server programs, we `wget` the homepages of top 100 websites by Alexa [5], which contain HTML, scripts and images. We then configured the servers to host them. After that, we ran ApacheBench [7] to iteratively access these web pages with various concurrencies (i.e., concurrent connections = 1, 64, and 256). In our experiments, we did not observe any error; worker processes of server programs did not crash; outputs with and without BUDDY (i.e., temporally) were always identical, and outgoing data of the buddy instances did not trigger any divergence (except one case). The only inconsistency case we observed was a false positive in OpenSSL. Specifically, the `SSLeay` implementation of OpenSSL uses an uninitialized buffer that may contain unspecified values as an additional source of entropy for pseudo random number generation. As a result, nonce in the SSL handshake will be different in the buddy instances. We argue that, since reading uninitialized memory is classified as a type of memory error [77], it is

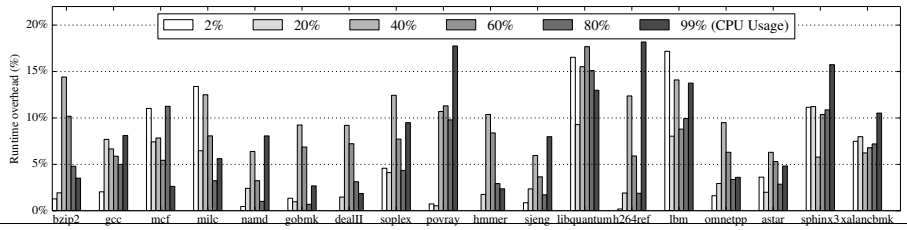

Figure 33: Performance of SPEC benchmarks when CPU is in various load levels. CPU usage is controlled by `stress-ng`. When CPU is highly-loaded (99% usage), BUDDY imposes an average performance overhead of 8.3%.

insecure and should be avoided at best[5]. In fact, OpenSSL developers are aware of this issue and have provided the `-DPURIFY` flag to avoid using uninitialized buffer as a source of randomness. After compiling OpenSSL with this flag, BUDDY functioned correctly without any false positive.

4.2.6.2 Security

To evaluate the security of our prototype implementation, we tested BUDDY with in-the-wild memory disclosure attacks. In this experiment, we enable both the partitioned address randomization scheme and the random padding scheme.

Direct address leak. Data-oriented exploits [82] showed that a format string vulnerability in Orzhttpd [6] could be exploited to leak the randomized addresses in the `GOTPLT` table. We reproduced the exploit in the same way— a data pointer used to retrieve the HTTP protocol version string is changed to point to the address of `GOTPLT` table. When the server responds to client, the content of `GOTPLT` table is sent to client. We applied BUDDY to Orzhttpd and tested the leak. The result shows that the leak is reliably detected when the randomized addresses in `GOTPLT` table are written into socket.

Crash-based address leak. Blind ROP [20] showed that ASLR can be bypassed within a few minutes. Its core idea is to leverage a simple stack buffer overflow vulnerability to

[5] Other SSL libraries, such as Google's BoringSSL, explicitly rely on system's randomness (e.g., `/dev/urandom`) [14]

[6] http://code.google.com/p/orzhttpd/source/detail?r=141

overwrite the return address with guessed value byte by byte, and based on whether the worker process crashes or not, it can get 1-bit information each time. We ran the BROP attack[7] against Nginx-1.4.0 with BUDDY applied. Again, the result shows that BUDDY is also able to detect Blind ROP attack because this attack cannot succeed in both buddy instances. When one instance reaches the point of socket write, the other one crashes, so BUDDY is able to detect this divergence.

Control-flow-based leaks. Information can be implicitly leaked through program control flow. However, as long as the leaked data is meaningful for the attacker to infer the randomized address, it must be different when outputted from the buddy instances even if it is implicitly transformed based on control flow. Seibert et al. [157] developed an attack that redirects a data pointer to a chosen byte by overwriting the pointer value. The data pointer is then dereferenced, and the chosen byte is used as the terminating condition of loop iteration. Therefore, the timing of executing the loop indirectly reflects the value of the chosen byte. We crafted this attack as in [157] and ran BUDDY against it. Since dereferencing the attacker-provided data pointer is an absolute address-based over-read, BUDDY is capable of detecting such an attack.

HeartBleed Leak. The over-read vulnerability (caused by `memcpy`) was exploited for the HeartBleed attack – leaking sensitive data. This attack can leak up to 64KB each time. To test BUDDY with this attack, we downloaded the vulnerable openssl-1.0.1a and obtained the existing exploit[8]. We then enforced the random padding scheme to this OpenSSL and ran the exploit against it. It turns out the exploit is always detected because the over-read always obtain different values.

[7] http://www.scs.stanford.edu/brop/
[8] https://www.exploit-db.com/exploits/32998/

Table 23: Performance (processing time in `ms` per request) of Apache httpd under BUDDY with various numbers of worker processes and workloads. `s=1MB`.

Level	c = 1		c = 16		c = 64		c = 256		
Worker	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Average**
p=1	16.2	17.2	15.2	16.8	14.5	16.1	13.6	14.8	9.2%
p=2	15.5	17.2	10.1	10.5	9.8	10.7	9.7	10.4	7.5%
p=3	16.2	16.6	9.7	9.7	9.5	9.7	9.4	9.7	1.7%
p=4	15.9	17.1	9.8	10.0	9.5	9.7	10.5	11.0	4.3%
p=5	17.1	17.4	9.4	9.5	9.7	9.8	10.6	11.2	2.0%
p=6	16.1	17.4	9.3	9.7	9.7	9.8	10.7	11.8	6.2%
p=7	15.2	16.7	9.2	9.5	9.8	10.1	11.2	12.3	6.5%
p=8	16.6	16.6	9.4	10.1	10.6	10.9	12.4	13.1	4.0%
Geomean		4.4%		2.1%		2.9%		6.2%	**3.6%**

Table 24: Performance (number of requests per second) of popular web servers with and without BUDDY under various workloads. `p=4`, `s=1MB`.

Level	c = 1		c = 16		c = 64		c = 256		
Server	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Average**
Apache	63.0	58.3	102.4	100.0	105.1	103.2	95.6	91.0	4.1%
Nginx	64.0	60.7	55.2	52.7	52.6	51.1	31.6	30.1	4.7%
Lighttpd	63.3	60.3	58.1	54.9	59.2	54.8	40.3	37.6	6.2%
PHP	64.9	64.7	58.7	54.2	44.5	42.9	28.2	26.9	4.0%
Geomean		2.9%		4.6%		3.4%		5.1%	**3.9%**

Table 25: Performance (rate of data transfer, in `MB/s`) of popular web servers with and without BUDDY when serving various sizes of resource. `p=4`, `c=16`.

Level	s = 1KB		s = 256KB		s = 1MB		s = 16MB		
Server	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Orig.**	**BUDDY**	**Average**
Apache	2.3	2.2	78.7	71.8	102.4	100.0	111.1	105.1	5.0%
Nginx	2.8	2.6	17.2	17.0	55.2	52.7	96.5	90.1	4.5%
Lighttpd	3.3	3.0	17.4	16.9	58.1	54.9	105.6	99.1	5.4%
PHP	2.4	2.2	17.3	16.4	58.7	54.2	93.1	91.3	5.5%
Geomean		5.4%		3.9%		4.7%		4.5%	**4.6%**

4.2.6.3 Performance of the BUDDY *Framework*

We first measured the performance of the BUDDY framework with only the partitioned address randomization scheme. In this evaluation, the random padding scheme is not included, which will be evaluated in §4.2.6.4. We conducted an extensive performance evaluations using both CPU intensive programs, represented by the SPEC2006 benchmarks;

and I/O intensive programs, represented by four popular server programs: Apache web server, Nginx web server, PHP, and Lighttpd. Note that all performance numbers are measured over *a pair of buddy instances*, not individual instance.

SPEC Benchmarks. We first tested the performance of BUDDY with the SPEC2006 benchmarks. We evaluated all C and C++ benchmarks and the results are the average number over 10 executions so as to minimize the effect of random fluctuations.

Table 26: Performance evaluation of BUDDY on the SPEC2006 benchmarks

Programs	**Baseline**	**BUDDY**	(%)	**+RandPad (%)**	
perlbench	3.53	3.64	(3.1%)	3.65	(3.4%)
bzip2	4.37	4.42	(1.1%)	4.42	(1.1%)
gcc	0.928	0.947	(2.0%)	0.979	(5.5%)
mcf	2.11	2.32	(10.0%)	2.39	(13.3%)
milc	4.03	5.03	(24.8%)	5.12	(27.0%)
namd	8.87	8.88	(0.1%)	8.93	(0.7%)
gobmk	13.2	13.6	(3.0%)	13.7	(3.8%)
dealII	10.7	10.8	(0.9%)	11.4	(6.5%)
soplex	0.242	0.288	(19.0%)	0.288	(19.0%)
povray	0.424	0.431	(1.7%)	0.441	(4.0%)
hmmer	2.00	2.01	(0.5%)	2.02	(1.0%)
sjeng	2.95	2.99	(1.4%)	3.09	(4.7%)
libquantum	0.0355	0.0399	(12.4%)	0.0413	(16.3%)
h264ref	9.30	9.33	(0.3%)	9.57	(2.9%)
lbm	1.69	1.93	(14.2%)	1.93	(14.2%)
omnetpp	0.307	0.308	(0.3%)	0.318	(3.6%)
astar	7.17	7.21	(0.6%)	7.32	(2.1%)
sphinx	1.06	1.13	(6.6%)	1.14	(7.5%)
xalancbmk	0.0564	0.0653	(15.8%)	0.0676	(19.9%)
Geomean	(s)	**2.34**%		**5.16**%	

Performance on lightly loaded system. The performance overhead (geo-mean) of BUDDY is 2.34% when there is no additional load on the system. Table 26 shows details. For comparison, CFI approaches typically impose 2-10% performance overhead and they do not protect attacks against data pointers. BUDDY is even slightly more efficient than Readactor [49] that protects code pointer only and requires source code and hardware assistance. BUDDY is significantly more efficient than previous n-version systems (e.g., even the latest n-version system Varan [81] imposes 14.2% performance overhead on SPEC benchmarks). We believe the performance gain is from the fact that BUDDY synchronizes

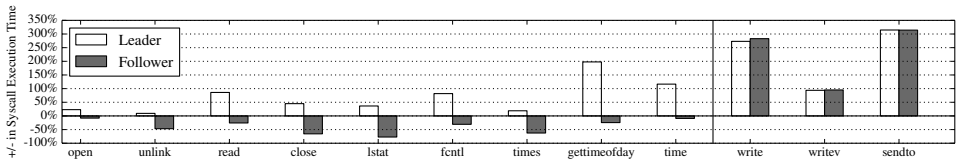

Figure 34: Micro-benchmarks on syscall execution. A negative overhead could only happen to the follower instance.

only I/O write.

Performance on heavily loaded system. The above experiment is performed on an unsaturated CPU. Since BUDDY double-executes the user-space code of target program, it is important to understand how BUDDY performs when the system is already heavily loaded. To get such performance results, we used `stress-ng`[9] to control the CPU usage so that it ranges from 2% (lowest load) to 99% (highest load, to finish the execution of benchmarks, we cannot set CPU usage to 100%). After setting the CPU usages, we ran SPEC benchmarks with and without BUDDY, and then computed the runtime overhead introduced by BUDDY. BUDDY imposes a runtime overhead of 8.3% when CPU is highly loaded (i.e., CPU usage=99%). The detailed results are shown in Figure 33.

The low overhead on SPEC benchmarks indicates that BUDDY is also a practical solution in protecting CPU intensive programs.

Web Benchmarks. Since we are focusing on remote attacks, we believe the performance of BUDDY for web server programs is more representative for practical deployment. For web servers, we simulate various scenarios by combining the following three parameters: (1) number of worker processes (denoted by `p`); (2) number of concurrent connections (denoted by `c`); (3) size of requested file (denoted by `s`).

Note that since the experiments are conducted on an eight-core machine, we evaluated the cases with number of worker processes from `p=1` to `p=8`. The size variable `s` has values of 1KB, 256KB, 1MB and 16MB, to simulate the cases when requested resource is metadata, HTML/Javascript file, images or large files, respectively. In terms of concurrency level, `c=1`

[9]http://kernel.ubuntu.com/ cking/stress-ng/

means the requests are always processed sequentially. Since web server is not CPU intensive, we used `c=256` to simulate a *saturated* load condition (i.e., when `c=256`, our Gigabit Ethernet I/O is already saturated). To reduce noise, experiment with each setting is repeated 10 times.

Given that many server programs use multi-processing/threading to boost performance as well as their capability to handle concurrent requests, our first experiment measured the performance of Apache httpd with different *numbers of worker processes* and different *numbers of concurrent connections*. Table 23 shows the average processing time of Apache httpd with different configurations. BUDDY introduces an overhead of 0.04% - 10.79%, which indicates that BUDDY is capable of handling multi-process programs efficiently. When server is in the simulated saturated condition (i.e., c=256), BUDDY introduces an average performance overhead of 6.83%.

To show that BUDDY can also be applied to other server programs, we evaluated the performance of BUDDY with different server programs, including Nginx (multi-processing server), Lighttpd, and PHP embedded server (single-threaded servers). In this experiment, we set the number of worker process to 4 (the default number by Apache) for both Apache httpd and Nginx. To measure the impact of different *concurrency level*, we fixed the size to 1MB. The result of this experiment is shown in Table 24, and the average overhead of BUDDY is less than 8%. To measure the impact of different *file size*, we set the concurrency to 16. Table 25 show the results and the average overhead of BUDDY is also less then 8%.

Microbenchmarks on Syscalls. We used `RDTSC` (interception of `RDTSC` and alert for divergence are temporarily disabled) to get the CPU cycles to accurately measure the cost of syscall virtualization. We tested the SPEC2006 benchmarks and server programs (Apache httpd, Nginx, Lighttpd and PHP embedded web server) to generate a profile of syscall performance. We measured the actual CPU cycles elapsed in (1) the syscall execution without BUDDY; (2) the syscall execution in leader instance; (3) the syscall execution in follower instance. Among all the executed syscalls, we selected 12 most popular syscalls (Figure 34). The first 9 syscalls are virtualization-only syscalls, hence are executed by leader

without waiting for follower's arrival; while the last 3 syscalls are synchronized. The results (normalized to the overhead over normal execution) showed that: (1) for virtualization-only syscalls, the execution time for the leader instance is longer (for updating the ring buffer) with an average overhead of 68%; but the execution time for the follower instance is often faster than normal execution with an average reduction of 39%. (2) kernel level virtualization is usually faster than virtualizing virtual syscalls like `gettimeofday` and `time` at user level. (3) for syscalls that needs synchronization, the execution time for both leader process and follower process increases, with average overhead of 227% and 231%, respectively.

Memory consumption. Since BUDDY double-executes the user-space code, the memory consumed by the user-space of the target program can be doubled. Code segments are the same in both buddy instances so can enjoy the Copy-on-Write mechanism. Therefore, only the memory of user-space data segments is doubled. Moreover, the instance controller also consumes some memory; however, its memory consumption is constant—independent on the memory usage of target program.

4.2.6.4 Performance of Random Padding Scheme

At last, we measured the performance of the random padding scheme. Since it only inserts instructions that will not trap into kernel, we use the CPU intensive SPEC benchmarks to test its performance overhead. All the test setting are the same as in §4.2.6.3, and the results are also shown in Table 26. In addition to the overhead introduced by the BUDDY framework, the random padding scheme further incurs a runtime overhead of 2.8%, which is small.

4.2.7 Limitations and Discussion

Unifying attacks. The effectiveness of BUDDY relies on that a memory disclosure attack will read divergent data at a detecting point because of diversification. One possible attack to circumvent BUDDY is unifying the diversification. Specifically, if attackers have the write primitive, they can overwrite the memory in one instance to make the value the same as the other, so that a memory disclosure can get the same value, breaking the divergence property.

Since attackers do not know the value in the other instance, the best attack strategy is to launch a probing attack: randomly overwriting some bytes in one instance and testing if the other has the same value (i.e., bypassing the divergence detection). If the values are not the same, BUDDY will detect the probing attack. Therefore, the success rate for such a unifying attack is $1/2^N$, where N is the number of bits to be disclosed.

Hardware resource consumption. The low performance overhead of BUDDY benefits a lot from multi-core processors. Since BUDDY double-executes user-space code, the performance overhead can be more than 100% on a single-core processor, which is an obvious limitation of BUDDY. As such, a multi-core processor is recommended to enjoy the security and efficiency provided by BUDDY. Given that most devices are equipped with processor with increased number of CPU and size of cache, we believe BUDDY is beneficial to users who give priority to security and privacy.

False positive. Reading uninitialized data may cause divergence at I/O write. We do not plan to handle uninitialized data read for two reasons. (1) We believe using uninitialized data is a bad programming practice and is a violation of memory safety [16, 77, 134]. (2) Leaking uninitialized data out is not always a "false positive", they may indeed leak addresses or sensitive data in many other cases [172]. On the other hand, if there is a real false positive due to some virtualizing points are not covered, it is straightforward to include them under the design of BUDDY.

Side-channel based leaks. In general, side-channel based leaks cannot be completely prevented because it is hard to identify *all* possible side-channels and leakage methods. The best we can do is to mitigate or detect existing or known side channel based attacks [26, 51]. In §4.2.6, we showed that BUDDY can successfully detection a timing side-channel attack [158] and a crashing side-channel attack [20]. Although BUDDY can prevent these specific side-channel attacks, we want to emphasize that BUDDY cannot prevent side-channel attacks in general.

Supporting script environments. The detection mechanism of BUDDY is general—two

Table 27: Comparison with representative n-version or multi-execution systems.

System	Synchronization		False	Overhead		Addr.	Over-	Defeat
	strict?	target	report	(SPEC)	(latency)	leak*	read*	ROP*
N-variant [45]	yes	all syscalls	high	N/A	17.8%	✗	✗	✓
Varan [81]	no	all syscalls	low	14.2%	2.4%	✗	✗	✗
DCL [182]	yes	most syscalls	high	6.37%	N/A	✗	✗	✓
TightLip [192]	yes	all syscalls	high	N/A	5%	✓	✗	✗
ShadowExe. [28]	yes	all inputs	high	N/A	>100%	✓	✗	✗
ReMon [183]	yes	some syscalls	low	3.1%	2.4%	✓	✗	✓
MvArmor [94]	yes	syscalls	low	9%	55%	✓	heap	✓
LDX [100]	no	outputs	high	4.7%	N/A	✓	✗	✓
Detile [70]	yes	interpreter	low	N/A	17%	✓	✗	✓
BUDDY	yes	I/O write	low	2.34%	4%	✓	✓	✓

instances are virtualized to act as a single one, and outgoing data is checked for divergence. The primary steps to implement a BUDDY system are to (1) clearly define the boundary between internal and external; (2) identify virtualizing points so that external behaviors of two instances (e.g., user interaction) can be virtualized to be performing as a single entity; (3) clearly define and monitor the leaking channels. So in theory, BUDDY can also be extended to support script environment, such as the JavaScript engine in browsers, and prevent attacks like JIT-ROP. The challenges, however, lie in the facts that the current architecture of browsers do not have a clear boundary and there are excessive leaking channels, especially in JIT (just-in-time compilation) mode. Specifically, because the JITed code is in the same process address space as the browser, it may read any memory address. To prevent attacks from scripts, we either need to disable JIT or check every memory read from JITed code; but either way will introduce a significant performance overhead.

4.2.8 Related Work

Address Protection. Readactor [49], Binary Stirring [187], and ASLR-Guard [120] prevent code address leak by either decoupling the code pointer and code address so that leaking the code pointers does not directly reveal the address of code, or hiding the code pointers by isolation. None of these approaches can protect data pointers that are protected by BUDDY. PointGuard [43] uses a single key to `XOR` all pointers, which is vulnerable to

chosen-plaintext attacks [120]. BUDDY does not suffer from these issues.

To mitigate the impact of address leak, many proposed approaches [12, 80, 92, 187] improved ASLR in granularity. These fine-grained randomizations have been shown to be vulnerable to memory disclosure attacks [167]. Several re-randomization approaches [19, 71, 116] have been proposed to improve ASLR in frequency. OS-ASR [71] requires kernel modification. TASR [19] does not re-randomizes data sections. A general problem with such re-randomization approaches is the significant performance overhead. Compared to these approaches, BUDDY is a general memory disclosure mitigation framework that can prevent leaks of both code and data addresses.

N-Version and Multi-Execution. Many systems leverage n-version or multi-execution techniques to achieve security or privacy goals [16, 27, 35, 101, 124, 152]. BUDDY leverages the same concept for a new goal—mitigating memory disclosures caused by memory errors. More importantly, BUDDY illustrates that with a single synchronization point, we can effectively detect data leaks with a small performance overhead. Further, we design the random padding scheme to enable BUDDY to generally detect data leaks caused by spatial memory errors. Specifically, we differentiate BUDDY from traditional n-version or multi-execution systems in some important aspects. In general, all traditional n-version systems *have to* synchronize all syscalls to achieve the security—they have to use lockstep to strictly synchronize both instances at each syscall; otherwise, attacks can succeed in the faster instance. Such design has two main issues: (1) performance overhead, waiting and notifying for lockstep at each syscall is expensive; and (2) false positives, some internal divergences are normal and should not be reported as leak. For example, compilers often inset paddings into data structures for alignment. These padding bytes are usually uninitialized thus may contain divergent values in the two buddy instances, which should not be treated as a leak if the uninitialized data do not leave the OS. Unlike traditional n-version systems, BUDDY reports divergence only at I/O write—when the uninitialized data indeed leaves the OS. This design choice minimizes the performance overhead and false positives. Table 27 summarizes

the major differences between BUDDY and representative n-version and multi-execution systems. More specifically, *n-variant systems* [45] imposes a significant performance overhead caused by the synchronization of every syscall. DCL [182] only focuses on code-reuse attacks without preventing data-only attacks [82]. Both approaches focus on preventing control flow hijacking attacks. In contrast, BUDDY prevents not only control flow hijacking attacks by protecting randomized address but also general data leaks (e.g., HeartBleed) caused by spatial memory errors. Once more diversifications are available, BUDDY can be easily adopted to enable more protections. Varan [81] is an efficient and general n-version execution framework. It however does not strictly synchronize syscalls, allowing the leader instance to leak data out without waiting for the follower to arrive (i.e., the asymmetrical attacks [182]). BUDDY does not suffer from such attacks, because I/O write is synchronized. TightLip [192] and Shadow execution [28] use multi-execution to prevent privacy leaks. In particular, TightLip triggers parallel execution only when pre-defined private data is accessed. It cannot support programs having control flow dependent on inputs. Shadow execution [28] achieves shadow execution at virtual-machine level, which dramatically enlarges the TCB and increases the performance overhead.

This year, concurrent works, ReMon [183], MvArmor [94], and LDX [100] also support selective synchronization. However, ReMon [183] still uses `ptrace` for syscall interception, whose performance can be improved with the syscall table-patching-based approach (Table 4.2.4.2) employed by BUDDY. MvArmor [94] cannot detect over-read resulted from stack, and its performance overhead is still significant. LDX [100] mutates inputs to detect divergence, which cannot detect over-read. In contrast, BUDDY can efficiently detect address leaks and general memory disclosures caused by out-of-bound over-read resulted from both stack and heap. Detile [70] instead detects address leaks in script environment. Since it has to instrument call and return bytecodes in interpreter, which are executed frequently, it imposes a performance overhead of 17%. Further, it does not detect general out-of-bound over-reads.

Memory Corruption Preventions. Memory errors can be exploited to leak data. Many memory safety approaches [16, 99, 129, 131, 132] provide a strong protection against a broad range of memory errors (e.g., out-of-bound read/write and use-after free). However, they usually require the source code or hardware support. BUDDY prevents memory disclosures without the requirements of source code or specific hardware features. Besides, memory safety approaches usually impose a significant performance overhead. For example, even the recent hardware-accelerated memory safety approach WatchdogLite [129] imposes a 29% performance overhead on the SPEC Benchmarks. More importantly, memory safety approaches cannot always prevent information leaks. For example, they usually do not prevent control-flow-based information leaks [158].

Quite a few control flow integrity (CFI) approaches [3, 123, 127, 136, 137, 178, 181, 193, 194] have been proposed to mitigate code reuse attacks assuming ASLR is bypassed. CFI can provide a strong protection even when address is leaked. Unfortunately, they are often shown to be bypassable [66, 72, 106, 178]. It is worth noting that Control-Flow Bending [30] illustrates that even fully-precise static CFI can be bypassed in some scenarios. Unlike CFI approaches, by preventing leaks of randomized addresses, BUDDY not only stops code reuse attacks but also data-oriented exploits (that hijack data pointers).

CHAPTER V

FUTURE WORK AND CONCLUSION

5.1 Future Work

We listed the causes of reported information leaks in Table 1. In this thesis, we have presented how to eliminate the most common information-leak vulnerabilities and fix some insecure designs that may cause information leaks. However, the table is by no means complete, and there are more causes of information leaks. In addition, we still do not have effective prevention against causes such as side channels.

Logic errors and design flaws. Information-leak vulnerabilities can be caused by memory errors, logic errors, and design flaws. In contrast to memory errors which have clear patterns, logic errors and design flaws are diverse, whose patterns are ad hoc. With the continuously introduced new features, software systems will suffer from a variety of unknown logic errors and design flaws. In general, we have to fix logic errors and design flaws case by case because we are not even able to abstract a uniform pattern for them. Fixing such vulnerabilities is therefore challenging and requires long-term research efforts. One challenge in this research direction is discovering new logic errors and design flaws. Since we do not have the pattern of unknown logic errors and design flaws, we plan to leverage empirically analyses (e.g., analyzing patch history of well-maintained software systems) and fuzzing techniques to discover new classes of vulnerabilities in systems.

Detecting (side-channel) probing attacks. A number of attacks [73, 104, 157] show that sensitive information in systems can be efficiently inferred by repeatedly probing victim systems. Traditional side-channel attacks are examples of probing attacks, which repeatedly perform certain operations and use statistical methods to infer secrete data such as encryption keys. While randomization-based techniques are increasingly used as a

building block of defense mechanisms [99, 118], recently probing attacks [73, 157] have been targeting derandomization to break these defense mechanisms. A general pattern of all these probing attacks is that attackers have to repeatedly and frequently probe the target system to infer secrete information. Such probing attacks will introduce abnormal features such as significant CPU cache miss. Therefore, by monitoring proper features, we can generally detect probing attacks. However, detection accuracy and efficiency are the main challenges. We will explore using multiple hardware features, including Intel Performance Counter Monitoring (PCM), Intel Cache Monitoring Technology (CMT), Intel Memory Bandwidth Monitoring (MBM), and Intel Process Trace (PT) for efficient monitoring. We will also design algorithms to synthesize these metrics to improve detection accuracy.

5.2 Conclusions

Vulnerabilities and insecure designs are very common in foundational software systems such as OS kernels and web servers. This is because that these systems are still implemented in unsafe programming languages, and system designers often prioritize performance over security. We have continuously witnessed critical system attacks exploiting vulnerabilities and insecure designs. Two typical goals of system attacks are to control victim systems and to leak the sensitive data. It is critical that a single system attack may cause a disaster to our cyber world, so it is pressing to prevent both data leaks and control attacks.

In this thesis, we proposed defeating both data leaks and control attacks by preventing information leaks. We identified that preventing information can be a general and efficient solution to defeating both attacks. We have worked on three ways to prevent information leaks: eliminating information-leak vulnerabilities, retrofitting system designs against information leaks, and protecting sensitive data from information leaks. For each way, we have developed multiple practical tools. Extensive evaluation results show that preventing information leaks can be achieved automatically, reliably, and efficiently, even for complex software systems.

REFERENCES

[1] "LLVM Classes Definition," 2016. http://llvm.org/docs/doxygen/html/annotated.html.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., and LIGATTI, J., "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Nov. 2005.

[3] ABADI, M., BUDIU, M., ERLINGSSON, Ú., and LIGATTI, J., "Control-flow integrity," in *ACM Conference on Computer and Communication Security*, 2005.

[4] AKRITIDIS, P., "Cling: A memory allocator to mitigate dangling pointers," in *Proceedings of the 19th USENIX Security Symposium (Security)*, (Washington, DC), Aug. 2010.

[5] ALEXA, "Alexa Top 500 Global Sites," 2016. http://www.alexa.com/topsites.

[6] ALEXA INTERNET, INC., "Top 500 Global Sites." http://www.alexa.com/topsites.

[7] APACHE, "ab - Apache HTTP server benchmarking tool," 2016. https://httpd.apache.org/docs/2.2/programs/ab.html.

[8] ARAVIND PRAKASHM XUNCHAO HU, H. Y., "vfguard: Strict protection for virtual function calls in cots c++ binaries," in *22st Annual Network and Distributed System Security Symposium*, 2015.

[9] ARIAS, O., DAVI, L., HANREICH, M., JIN, Y., KOEBERL, P., PAUL, D., SADEGHI, A.-R., and SULLIVAN, D., "Hafix: Hardware-assisted flow integrity extension," in *52nd Design Automation Conference (DAC)*, 2015.

[10] ATHANASAKIS, M., ATHANASOPOULOS, E., POLYCHRONAKIS, M., PORTOKALIDIS, G., and IOANNIDIS, S., "The devil is in the constants: Bypassing defenses in browser jit engines," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2015.

[11] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., and PEWNY, J., "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 21st ACM conference on Computer and communications security*, 2014.

[12] BACKES, M. and NÜRNBERGER, S., "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd USENIX Security Symposium*, Aug. 2014.

[13] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., and USTUNER, A., "Thorough static analysis of device drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, 2006.

[14] BENJAMIN, D., "BoringSSL," Oct. 2015. `https://www.imperialviolet.org/2015/10/17/boringssl.html`.

[15] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., and GROSSMAN, D., "Coredet: A compiler and runtime system for deterministic multithreaded execution," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[16] BERGER, E. D. and ZORN, B. G., "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Ottawa, Canada), June 2006.

[17] BERGER, E. D. and ZORN, B. G., "Diehard: probabilistic memory safety for unsafe languages," in *Acm sigplan notices*, vol. 41, pp. 158–168, ACM, 2006.

[18] BHATKAR, S. and SEKAR, R., "Data space randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, 2008.

[19] BIGELOW, D., HOBSON, T., RUDD, R., STREILEIN, W., and OKHRAVI, H., "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*, Oct 2015.

[20] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., and BONEH, D., "Hacking blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), 2014.

[21] BOJINOV, H., BONEH, D., CANNINGS, R., and MALCHEV, I., "Address space randomization for mobile devices," in *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, 2011.

[22] BOSMAN, E., RAZAVI, K., BOS, H., , and GIUFFRIDA, C., "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, (San Jose, CA, USA), IEEE, May 2016.

[23] BOSMAN, E., SLOWINSKA, A., and BOS, H., "Minemu: The world's fastest taint tracker," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, 2011.

[24] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., and SADEGHI, A.-R., "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks," in *Proceedings of the 26th USENIX Security Symposium (Security)*, (Vancouver, Canada), Aug. 2017.

[25] BRUENING, D. and ZHAO, Q., "Practical memory checking with dr. memory," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, (Washington, DC), Mar. 2011.

[26] BRUMLEY, B. B. and TUVERI, N., "Remote timing attacks are still practical," in *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, 2011.

[27] BRUSCHI, D., CAVALLARO, L., and LANZI, A., "Diversified process replica for defeating memory error exploits," *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 2007.

[28] CAPIZZI, R., LONGO, A., VENKATAKRISHNAN, V. N., and SISTLA, A. P., "Preventing information leaks through shadow executions," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, 2008.

[29] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., and JIANG, X., "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, (Chicago, IL), Nov. 2009.

[30] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., and GROSS, T. R., "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[31] CARLINI, N. and WAGNER, D., "ROP is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium*, 2014.

[32] CHEN, "Hey Man, Have You Forgotten To Initialize Your Memory?," 2015.

[33] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., and KAASHOEK, M. F., "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, (Shanghai, China), July 2011.

[34] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., and KAASHOEK, M. F., "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, p. 5, ACM, 2011.

[35] CHEN, L. and AVIZIENIS, A., "N-version programminc: A fault-tolerance approach to rellablllty of software operatlon," in *International Symposium on Fault-Tolerant Computing*, 1995.

[36] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., and IYER, R. K., "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium (Security)*, (Baltimore, MD), Aug. 2005.

[37] CHEN, X., SLOWINSKA, A., ANDRIESSE, D., BOS, H., and GIUFFRIDA, C., "StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2015.

[38] CHENG, Y., ZHOU, Z., YU, M., DING, X., and DENG, R. H., "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium*, 2014.

[39] CHOW, J., PFAFF, B., GARFINKEL, T., and ROSENBLUM, M., "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proceedings of the 14th Conference on USENIX Security Symposium*, (Berkeley, CA, USA), 2005.

[40] CLAUSE, J., DOUDALIS, I., ORSO, A., and PRVULOVIC, M., "Effective memory protection using dynamic tainting," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, 2007.

[41] COOK, K., "Kernel Exploitation Via Uninitialized Stack," 2011. `https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf`.

[42] COOK, K., "Kernel address space layout randomization," 2013. `http://outflux.net/slides/2013/lss/kaslr.pdf`.

[43] COWAN, C., BEATTIE, S., JOHANSEN, J., and WAGLE, P., "PointGuard TM: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[44] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., and ZHANG, Q., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium (Security)*, (San Antonio, TX), Jan. 1998.

[45] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., and HISER, J., "N-variant systems: A secretless framework for security through diversity," in *In Proceedings of the 15th USENIX Security Symposium*, 2006.

[46] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., and FRANZ, M., "Thwarting cache side-channel attacks through dynamic software diversity," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[47] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., and FRANZ, M., "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*, pp. 763–780, IEEE, 2015.

[48] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., and FRANZ, M., "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy*, 2015.

[49] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., and FRANZ, M., "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy*, 2015.

[50] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., and FRANZ, M., "It's a TRAP: Table randomization and protection against function reuse attacks," in *Proceedings of the 22nd ACM conference on Computer and communications security*, CCS '15, 2015.

[51] CROSBY, S. A., WALLACH, D. S., and RIEDI, R. H., "Opportunities and limits of remote timing attacks," *ACM Trans. Inf. Syst. Secur.*, Jan. 2009.

[52] CUI, H., SIMSA, J., LIN, Y.-H., LI, H., BLUM, B., XU, X., YANG, J., GIBSON, G. A., and BRYANT, R. E., "Parrot: A practical runtime for deterministic, stable, and reliable threads," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[53] DALTON, M., KANNAN, H., and KOZYRAKIS, C., "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007.

[54] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., and MONROSE, F., "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[55] DAVI, L., SADEGHI, A.-R., LEHMANN, D., and MONROSE, F., "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium*, 2014.

[56] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., and SADEGHI, A.-R., "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *8th ACM SIGSAC symposium on Information, computer and communications security (ACM ASIACCS 2013)*, pp. 299–310, ACM, 2013.

[57] DETAILS, C., "Vulnerabilities By Type," 2016. `http://www.cvedetails.com/vulnerabilities-by-types.php`.

[58] DHURJATI, D., KOWSHIK, S., and ADVE, V., "Safecode: enforcing alias analysis for weakly typed languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Ottawa, Canada), June 2006.

[59] DILLIG, I., DILLIG, T., and AIKEN, A., "Reasoning about the unknown in static analysis," *Commun. ACM*, 2010.

[60] DING, Y., WEI, T., WANG, T., LIANG, Z., and ZOU, W., "Heap taichi: exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 327–336, ACM, 2010.

[61] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., and WITCHEL, E., "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Hollywood, CA), Oct. 2012.

[62] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., and PAXSON, V., "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, 2014.

[63] EGELE, M., WURZINGER, P., KRUEGEL, C., and KIRDA, E., "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 88–106, Springer, 2009.

[64] ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., and OLYNYK, K., "Effective data-race detection for the kernel," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Vancouver, Canada), Oct. 2010.

[65] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., and OKHRAVI, H., "Missing the point(er): On the effectiveness of code pointer integrity," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[66] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., and SIDIROGLOU-DOUSKOS, S., "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*, Oct 2015.

[67] FLAKE, H., "Attacks on Uninitialized Local Variables," 2006. `http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf`.

[68] FRANTZEN, M. and SHUEY, M., "Stackghost: Hardware facilitated stack protection.," in *USENIX Security Symposium*, 2001.

[69] GAO, D., REITER, M. K., and SONG, D., "Behavioral distance for intrusion detection," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, 2006.

[70] GAWLIK, R., KOPPE, P., KOLLENDA, B., PAWLOWSKI, A., GARMANY, B., and HOLZ, T., "Detile: Fine-grained information leak detection in script engines," in *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, (San Sebastián, Spain), July 2016.

[71] GIUFFRIDA, C., KUIJSTEN, A., and TANENBAUM, A. S., "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[72] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., and PORTOKALIDIS, G., "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[73] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., and GIUFFRIDA, C., "ASLR on the Line: Practical Cache Attacks on the MMU," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2017.

[74] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., and CHENEY, J., "Region-based memory management in cyclone," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Berlin, Germany), June 2002.

[75] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., and FELTEN, E. W., "Lest we remember: cold-boot attacks on encryption keys," in *Proceedings of the 18th USENIX Security Symposium (Security)*, (Montreal, Canada), Aug. 2009.

[76] HARDEKOPF, B. and LIN, C., "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (San Diego, CA), June 2007.

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

[78] HERLANDS, W., HOBSON, T., and DONOVAN, P. J., "Effective entropy: Security-centric metric for memory randomization techniques," in *Proceedings of the 7th USENIX Conference on Cyber Security Experimentation and Test*, CSET'14, 2014.

[79] HERLANDS, W., HOBSON, T., and DONOVAN, P. J., "Effective entropy: Security-centric metric for memory randomization techniques," in *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, 2014.

[80] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., and DAVIDSON, J. W., "ILR: Where'd my gadgets go?," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[81] HOSEK, P. and CADAR, C., "Varan the unbelievable: An efficient n-version execution framework," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[82] HU, H., CHUA, Z. L., ADRIAN, S., SAXENA, P., and LIANG, Z., "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium (Security)*, (Washington, DC), Aug. 2015.

[83] INTEL, "Intel 64 and ia-32 architectures software developer's manual," 2014.

[84] INTEL, *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.* Intel Corporation, 2025.

[85] "J00RU" JURCZYK, M., "The story of CVE-2011-2018 exploitation," 2012. [Online; accessed 16-Aug-2016].

[86] JANG, D., TATLOCK, Z., and LERNER, S., "Safedispatch: Securing C++ virtual calls from memory corruption attacks," in *21st Annual Network and Distributed System Security Symposium*, 2014.

[87] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., and PORTOKALIDIS, G., "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 2013.

[88] JOHNSON, R. and WAGNER, D., "Finding user/kernel pointer bugs with type inference," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

[89] KEMERLIS, V. P., POLYCHRONAKIS, M., and KEROMYTIS, A. D., "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, (San Diego, CA), Aug. 2014.

[90] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., and KEROMYTIS, A. D., "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[91] KEMERLIS, V. P., PORTOKALIDIS, G., and KEROMYTIS, A. D., "kguard: Lightweight kernel protection against return-to-user attacks," in *Proceedings of the 21st USENIX Security Symposium (Security)*, (Bellevue, WA), Aug. 2012.

[92] KIL, C., JUN, J., BOOKHOLT, C., XU, J., and NING, P., "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22Nd Annual Computer Security Applications Conference*, 2006.

[93] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., and MUTLU, O., "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), 2014.

[94] KONING, K., BOS, H., and GIUFFRIDA, C., "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *Proceedings of the 46th International Conference on Dependable Systems and Networks (DSN)*, (Toulouse, France), June 2016.

[95] KRAUSE, M., "CVE-2013-1825: various info leaks in Linux kernel," 2013. `http://www.openwall.com/lists/oss-security/2013/03/07/2`.

[96] KURMUS, A. and ZIPPEL, R., "A tale of two kernels: Towards ending kernel hardening wars with split kernel," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, (Scottsdale, Arizona), Nov. 2014.

[97] KURMUS, A. and ZIPPEL, R., "A tale of two kernels: Towards ending kernel hardening wars with split kernel," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1366–1377, ACM, 2014.

[98] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., and SONG, D., "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[99] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., and SONG, D., "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[100] KWON, Y., KIM, D., SUMNER, W. N., KIM, K., SALTAFORMAGGIO, B., ZHANG, X., and XU, D., "Ldx: Causality inference by lightweight dual execution," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Atlanta, GA), Apr. 2016.

[101] LARSEN, P., HOMESCU, A., BRUNTHALER, S., and FRANZ, M., "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[102] LEE, B., LU, L., WANG, T., KIM, T., and LEE, W., "From Zygote to Morula: Fortifying weakened aslr on android," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[103] LEE, J., HAM, H., KIM, I., and SONG, J., "Poster: Page table manipulation attack," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1644–1646, ACM, 2015.

[104] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., and PEINADO, M., "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing (to appear)," in *Proceedings of the 26th USENIX Security Symposium (Security)*, (Vancouver, Canada), Aug. 2017.

[105] LI, L., JUST, J. E., and SEKAR, R., "Address-space randomization for windows systems," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, 2006.

[106] LIEBCHEN, C., NEGRO, M., LARSEN, P., DAVI, L., SADEGHI, A.-R., CRANE, S., QUNAIBIT, M., FRANZ, M., and CONTI, M., "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *22nd ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[107] LIN, Z., RILEY, R. D., and XU, D., "Polymorphing software by randomizing data structure layout," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, 2009.

[108] LIN, Z., ZHANG, X., and XU, D., "Automatic reverse engineering of data structures from binary execution," in *17th Annual Network and Distributed System Security Symposium*, 2010.

[109] LIU, T., CURTSINGER, C., and BERGER, E. D., "Dthreads: Efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[110] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., and XIA, Y., "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, (Denver, Colorado), Oct. 2015.

[111] LLVM, "LLVM Alias Analysis Infrastructure," 2016. `http://llvm.org/docs/AliasAnalysis.html`.

[112] LLVM, "The LLVM Compiler Infrastructure," 2016. `http://llvm.org/`.

[113] LLVMLINUX, "The LLVMLinux Project," 2016. `http://llvm.linuxfoundation.org/index.php/Main_Page`.

[114] LONG LE, "Exploiting nginx chunked overflow bug, the undisclosed attack vector." `http://ropshell.com/slides/Nginx_chunked_overflow_the_undisclosed_attack_vector.pdf`.

[115] LU, K., ZHOU, X., BERGAN, T., and WANG, X., "Efficient deterministic multithreading without global barriers," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014.

[116] LU, K., NÜRNBERGER, S., BACKES, M., and LEE, W., "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2016.

[117] LU, K., SONG, C., KIM, T., and LEE, W., "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, (Vienna, Austria), October 2016.

[118] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., and LEE, W., "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, (Denver, Colorado), Oct. 2015.

[119] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., and LEE, W., "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM conference on Computer and communications security*, CCS '15, 2015.

[120] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., and LEE, W., "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, (Denver, Colorado), Oct. 2015.

[121] LU, K., WALTER, M.-T., PFAFF, D., NÜRNBERGER, S., LEE, W., and BACKES, M., "Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying (to appear)," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2017.

[122] LU, K., XU, M., SONG, C., KIM, T., and LEE, W., "Preventing Data Leaks Using Replicated Execution," In submission.

[123] MASHTIZADEH, A. J., BITTAU, A., MAZIERES, D., , and BONEH, D., "Cryptographically enforced control flow integrity," 2014. arXiv preprint arXiv:1408.1451.

[124] MAURER, M. and BRUMLEY, D., "Tachyon: Tandem execution for efficient live patch testing," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[125] MCVOY, L. W. and STAELIN, C., "Lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996.

[126] MIN, C., KASHYAP, S., LEE, B., SONG, C., and KIM, T., "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, (Monterey, CA), Oct. 2015.

[127] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., and FRANZ, M., "Opaque control-flow integrity," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[128] MUSLINER, D. J., FRIEDMAN, S. E., BOLDT, M., BENTON, J., SCHUCHARD, M., KELLER, P., and MCCAMANT, S., "Fuzzbomb: Autonomous cyber vulnerability detection and repair," in *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV 2015)*, 2015.

[129] NAGARAKATTE, S., MARTIN, M. M. K., and ZDANCEWIC, S., "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, (Orlando, FL), Feb. 2014.

[130] NAGARAKATTE, S., MARTIN, M. M., and ZDANCEWIC, S., "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 189–200, IEEE Computer Society, 2012.

[131] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., and ZDANCEWIC, S., "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Dublin, Ireland), June 2009.

[132] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., and ZDANCEWIC, S., "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management*, 2010.

[133] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., and WEIMER, W., "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, May 2005.

[134] NETHERCOTE, N. and SEWARD, J., "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (San Diego, CA), June 2007.

[135] NIU, B. and TAN, G., "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[136] NIU, B. and TAN, G., "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[137] NIU, B. and TAN, G., "Per-input control-flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[138] NOSSUM, V., "Getting Started With kmemcheck," 2015. `https://www.kernel.org/doc/Documentation/kmemcheck.txt`.

[139] NOVARK, G. and BERGER, E. D., "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 573–584, ACM, 2010.

[140] OLSZEWSKI, M., ANSEL, J., and AMARASINGHE, S., "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Washington, DC), Mar. 2009.

[141] ÖZKAN, S., "CVE Details: Linux kernel security vulnerabilities - overview," 2016. [Online; accessed 15-Aug-2016].

[142] PAPPAS, V., POLYCHRONAKIS, M., and KEROMYTIS, A. D., "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[143] PAPPAS, V., POLYCHRONAKIS, M., and KEROMYTIS, A. D., "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[144] PAX TEAM. `http://pax.grsecurity.net/`.

[145] PAX TEAM, "PaX Address Space Layout Randomization (ASLR)." `http://pax.grsecurity.net/docs/aslr.txt`.

[146] PAX TEAM, "PaX Address Space Layout Randomization (ASLR)." `http://pax.grsecurity.net/docs/aslr.txt`.

[147] PAYER, M., *HexPADS: A Platform to Detect "Stealth" Attacks*, pp. 138–154. Cham: Springer International Publishing, 2016.

[148] PEIRÓ, S., NOZ, M. M., MASMANO, M., and CRESPO, A., "Detecting stack based kernel information leaks," in *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, 2014.

[149] PRASAD, M. and CKER CHIUEH, T., "A binary rewriting defense against stack based overflow attacks," in *In Proceedings of the USENIX Annual Technical Conference*, 2003.

[150] RATANAWORABHAN, P., LIVSHITS, V. B., and ZORN, B. G., "Nozzle: A defense against heap-spraying code injection attacks.," in *USENIX Security Symposium*, pp. 169–186, 2009.

[151] RENTZSCH, J., "Data alignment: Straighten up and fly right – Align your data for speed and correctness," 2005. `https://www.ibm.com/developerworks/library/pa-dalign/pa-dalign-pdf.pdf`.

[152] SALAMAT, B., JACKSON, T., GAL, A., and FRANZ, M., "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, 2009.

[153] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., and HOLZ, T., "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *36th IEEE Symposium on Security and Privacy*, 2015.

[154] SCHWARTZ, E. J., AVGERINOS, T., and BRUMLEY, D., "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[155] SCHWARTZ, E. J., AVGERINOS, T., and BRUMLEY, D., "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[156] SEABORN, M. and DULLIEN, T., "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[157] SEIBERT, J., OKHRAVI, H., and SÖDERSTRÖM, E., "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[158] SEIBERT, J., OKHRAVI, H., and SODERSTROM, E., "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[159] SEIBERT, J., OKKHRAVI, H., and SÖDERSTRÖM, E., "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 54–65, 2014.

[160] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., and VYUKOV, D., "Address-Sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, (Boston, MA), June 2012.

[161] SERNA, F. J., "The info leak era on software exploitation," 2012. Blackhat USA.

[162] SEWARD, J. and NETHERCOTE, N., "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, (Anaheim, CA), June–July 2005.

[163] SEWARD, J. and NETHERCOTE, N., "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[164] SHACHAM, H., "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Oct.–Nov. 2007.

[165] SHACHAM, H., "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[166] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., and BONEH, D., "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[167] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., and SADEGHI, A.-R., "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, (San Francisco, CA), May 2013.

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

[169] SONG, C., LEE, B., LU, K., HARRIS, W. R., KIM, T., and LEE, W., "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2016.

[170] SONG, C., ZHANG, C., WANG, T., LEE, W., and MELSKI, D., "Exploiting and protecting dynamic code generation.," in *NDSS*, 2015.

[171] SONG, J. and ALVES-FOSS, J., "The darpa cyber grand challenge: A competitor's perspective," *IEEE Security & Privacy*, vol. 13, no. 6, pp. 72–76, 2015.

[172] STEPANOV, E. and SEREBRYANY, K., "MemorySanitizer: fast detector of uninitialized memory use in C++," in *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, (San Francisco, CA), Feb. 2015.

[173] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., and VIGNA, G., "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[174] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., and WALTER, T., "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009.

[175] TEAM, P., "PaX - gcc plugins galore," 2013. https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf.

[176] THANASSIS, H. A., KIL, C. S., and DAVID, B., "Aeg: Automatic exploit generation," in *ser. Network and Distributed System Security Symposium*, 2011.

[177] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., and PIKE, G., "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium*, 2014.

[178] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., and PIKE, G., "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium*, 2014.

[179] TILAK, S., HUBBARD, P., MILLER, M., and FOUNTAIN, T., "The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems," in *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '07, 2007.

[180] TORVALDS, L., "Linux Kernel Git Repository," 2016. [Online; accessed 5-Aug-2016].

[181] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., and GIUFFRIDA, C., "Practical context-sensitive cfi," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[182] VOLCKAERT, S., COPPENS, B., and DE SUTTER, B., "Cloning your gadgets: Complete rop attack immunity with multi-variant execution," *IEEE Transactions on Dependable and Secure Computing*, 2015.

[183] VOLCKAERT, S., COPPENS, B., VOULIMENEAS, A., HOMESCU, A., LARSEN, P., SUTTER, B. D., and FRANZ, M., "Secure and efficient application monitoring and replication," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, (Denver, CO), June 2016.

[184] WANG, S., WANG, P., and WU, D., "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[185] WANG, X., CHEN, H., CHEUNG, A., JIA, Z., ZELDOVICH, N., and KAASHOEK, M. F., "Undefined behavior: What happened to my code?," in *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, (Seoul, South Korea), July 2012.

[186] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., and KAASHOEK, M. F., "Improving Integer Security for Systems with KINT," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Hollywood, CA), Oct. 2012.

[187] WARTELL, R., MOHAN, V., HAMLEN, K. W., and LIN, Z., "Binary Stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[188] WEI, T., WANG, T., DUAN, L., and LUO, J., "Secure dynamic code generation against spraying," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 738–740, ACM, 2010.

[189] XU, H. and CHAPIN, S., "Address-space layout randomization using code islands," in *Journal of Computer Security*, IOS Press, 2009.

[190] XU, W., LI, J., SHU, J., YANG, W., XIE, T., ZHANG, Y., and GU, D., "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 414–425, ACM, 2015.

[191] YE, D., SUI, Y., and XUE, J., "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, (Orlando, FL), Feb. 2014.

[192] YUMEREFENDI, A. R., MICKLE, B., and COX, L. P., "Tightlip: Keeping applications from spilling the beans," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, 2007.

[193] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., and ZOU, W., "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[194] ZHANG, M. and SEKAR, R., "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium*, 2013.

[195] ZHOU, Z., REITER, M. K., and ZHANG, Y., "A software approach to defeating side channels in last-level caches," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, (Vienna, Austria), Oct. 2016.