2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

What You See is Not What You Get! Thwarting Just-in-Time ROP with Chameleon

Ping Chen*, Jun Xu* , Zhisheng Hu†, Xinyu Xing*, Minghui Zhu†, Bing Mao‡, Peng Liu*
*College of Information Sciences and Technology, The Pennsylvania State University
†School of Electrical Engineering and Computer Science,Pennsylvania State University
‡State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
{pzc10,jxx13,xxing,pliu}@ist.psu.edu, {zxh128, mzu16}@psu.edu, maobing@nju.edu.cn

***Abstract*—Address space randomization has long been used for counteracting code reuse attacks, ranging from conventional ROP to sophisticated Just-in-Time ROP. At the high level, it shuffles program code in memory and thus prevents malicious ROP payload from performing arbitrary operations. While effective in mitigating attacks, existing randomization mechanisms are impractical for real-world applications and systems, especially considering the significant performance overhead and potential program corruption incurred by their implementation.**

In this paper, we introduce CHAMELEON, a practical defense mechanism that hinders code reuse attacks, particularly Just-in-Time ROP attacks. Technically speaking, CHAMELEON instruments program code, randomly shuffles code page addresses and minimizes the attack surface exposed to adversaries. While this defense mechanism follows in the footprints of address space randomization, our design principle focuses on using randomization to obstruct code page disclosure, making the ensuing attacks infeasible. We implemented a prototype of CHAMELEON on Linux operating system and extensively experimented it in different settings. Our theoretical and empirical evaluation indicates the effectiveness and efficiency of CHAMELEON in thwarting Just-in-Time ROP attacks.

***Keywords*—Address space randomization, JIT-ROP, OS kernel, binary instrumentation**

I. INTRODUCTION

With the broad deployment of W⊕X and Data Execution Prevention (DEP) [1], [2], the effectiveness of simple code injection attacks like worm wanes. In its place rises a more sophisticated attack, Return-Oriented Programming (ROP) [3], [4], [5], [6]. An ROP attack hijacks a target machine by gaining control of the call stack and executing carefully chosen machine instruction sequences (i.e., "gadgets"). This attack exploits the fact that the gadget locations in memory are deterministic and an attacker can easily chain them to perform arbitrary operations.

Address space randomization is a technique that obfuscates the locations of gadgets by randomizing the memory layout of a process. As such, it has been shown to be effective for the defense against ROP [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. However, recent research indicates this technique is vulnerable to Just-In-Time ROP (JIT-ROP) [17] in which an attacker employs memory disclosure attacks [18], [19] to locate gadgets in memory and thus circumvents address randomization.

To counteract JIT-ROP, existing defense mechanisms either restrict memory execution [20], [21], [22], [23] or perform adaptive randomization [24], [25], [26]. To some extent, these solutions are effective in terms of obstructing JIT-ROP. However, they are impractical for real-world applications and systems. From a usability standpoint, memory execution restriction requires either special hardware support [20] or extensive system modification [22]. For example, the implementation of Readactor [20] relies upon a processor with the support of Extended Page Table (EPT). In addition, memory execution restriction violates the design principle of modern processors, especially considering modern processors usually set an executable memory space readable while memory restriction defense typically sets a memory executable but not readable. From a correctness standpoint, adaptive randomization techniques do not guarantee program correctness especially when dealing with binary code. For example, Remix [25] and TASR [24] need to locate code pointers for randomization. However, a prior study has demonstrated that it is generally impossible to correctly locate all code pointers in memory [27]. From a performance standpoint, adaptive randomization techniques like Isomeron [26] and RuntimeASLR [27] perform dynamic code instrumentation which introduces significant overhead at run time.

In this paper, we propose CHAMELEON, a novel practical mechanism to counteract JIT-ROP. Our basic idea is to obfuscate the address of code pages, making code exploit infeasible. In JIT-ROP, an attacker typically exploits memory disclosure vulnerability, and thus obtains the address of code pages and corresponding code instructions. Using these information, the attacker then constructs ROP payload on call stack and performs arbitrary operations. In this work, we randomize the address of code pages through an address space transformation mechanism. In this way, the attacker is no longer able to obtain the actual address of a code page, dump the code instructions and thus construct a valid ROP payload. Technically speaking, CHAMELEON draws on address randomization. However, it is fundamentally different from existing address space randomization mechanisms. In fact, existing randomization techniques do not hinder code disclosure but impede malicious ROP payload

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.47

to perform arbitrary operations. In contrast, our randomization mechanism serves as the first-line defense that counteracts attacks by obstructing the code page disclosure directly.

More specifically, CHAMELEON constructs an address table that maps each code page address to a surrogate address right after the OS kernel initializes its page table. Then, it substitutes each code page address with the corresponding surrogate address in the page table. Considering such address transformation can destroy program execution, CHAMELEON further replaces certain instructions and instruments binary to ensure its correctness. In particular, CHAMELEON replaces instructions that transfer program control flow as well as the last instruction of each page because they may perform cross-page operations that need the actual virtual address.

Compared with the existing defense mechanisms (e.g., [21], [23]), CHAMELEON does not rely upon the basis of custom hypervisor/software emulation nor hardware assistant. It is designed in the kernel level of an operating system, and all its operations are implemented as a software solution. In addition, CHAMELEON introduces low overhead because it does not perform instrumentation at run time. Last but not least, the design of CHAMELEON guarantees the correctness of the program execution. Some existing ROP defense needs to perform binary instrumentation and update code pointers, which has been shown to be infeasible especially when there is no access to the source code. In our design, CHAMELEON only instruments a subset of code instructions and does not make any changes to code pointers.

In summary, the paper makes the following contributions.

- We designed CHAMELEON, a new practical defense mechanism. We demonstrated it can be easily deployed on modern operating systems and counteracts JIT-ROP attacks.
- We implemented a CHAMELEON prototype that takes a transformation of virtual addresses in a page table, statically instruments a program and ensures the correctness of the program execution.
- We evaluated CHAMELEON theoretically and empirically, and showed that it could effectively thwart JIT-ROP attacks with a moderate performance overhead.

The rest of the paper is organized as follows. Section II discusses some technical background. Section III presents the threat model and CHAMELEON overview. Section IV and V describe the design and implementation of CHAMELEON in detail followed by its theoretical and empirical evaluation in Section VI. Section VII categorizes ROP attacks and discusses the effectiveness provided by our defense under different types of attacks. Finally, the work is concluded in Section VIII.

II. BACKGROUND

In this section, we briefly describe some technical background. In particular, the section begins with the technical background of JIT-ROP attacks. Then, it summarizes and discusses some existing defense mechanisms.

A. JIT-ROP Attacks

A JIT-ROP attack is typically launched in two steps – *code reading* and *code using*. In code reading step, an attacker first exploits memory disclosure vulnerabilities and obtains the information about code pages. Then, he dumps the code pages, disassembles them and identifies useful code gadgets. Based on the knowledge from the first step, in the code using step, the attacker constructs ROP payload and manipulates the call stack accordingly. To illustrate this attack, Figure 1 presents a running example.

As is shown in Figure 1 (a) and (b), the attacker first exploits an overread vulnerability with a leaked virtual address (`0x5000`), and dumps the code page. Then, he disassembles the code page and identifies two gadgets starting at `0x5012` and `0x5096` respectively. In the code using step, the attacker manipulates call stack by using payload [`0x5012, 0x00000001, 0x5096`], where `0x5012` and `0x5096` represent the addresses of the two gadgets and `0x00000001` is the data used by the first gadget. At run time when stack pointer `%esp` points to the cell with `0x5012` and PC register `%eip` arrives at `0x5136`, the program execution is compromised and redirected to the gadgets chosen by the attacker.

B. Existing Defense Mechanisms

JIT-ROP is one type of code reuse attacks [3], [4], [28], [29], [6]. Over the past decade, a variety of defense mechanisms have been proposed to mitigate the effect of code reuse attacks. Here, we summarize and discuss these existing defense mechanisms in turn.

Control Flow Integrity. Control Flow Integrity (CFI) is a defense mechanism that prevents code reuse attacks from arbitrarily controlling program behavior [30], [31], [32], [33], [34], [35]. In particular, it confines all control flow transfers to follow a valid path defined in a static control flow graph. Theoretically, CFI can thwart many types of control-flow hijacking attacks, ranging from trivial ROP/JOP to sophisticated ROP (like JIT-ROP and BROP [36]). Although CFI has been proposed for many years, it has not been shown to be a practical solution in the real world. As is demonstrated in [37], [38], fine-grained CFI provides strong security guarantee but at the same time introduces performance overhead significantly high. While recent research indicates coarse-grained CFI can significantly improve performance, it also shows CFI is vulnerable to sophisticated attacks [39], [40], [41], [34].

Address and Code Randomization. Code reuse attack exploits the fact that an attacker can obtain information about the code in memory. To defend against this attack, a large amount of research therefore aims to make memory layout nondeterministic. In particular, prior research focuses on developing a variety of mechanisms to randomize the memory address of a program. For example, several existing work (e.g., [7], [8], [11], [13]) performs randomization at segment offset level, code page level and function level, respectively. In addition to memory randomization, some existing work proposes to perform randomization on code, e.g., code basic blocks [12] and code instructions [10], [14]. Code randomization can also take place upon code compiling (e.g., [14]) and code loading (e.g., [7], [8], [9]).

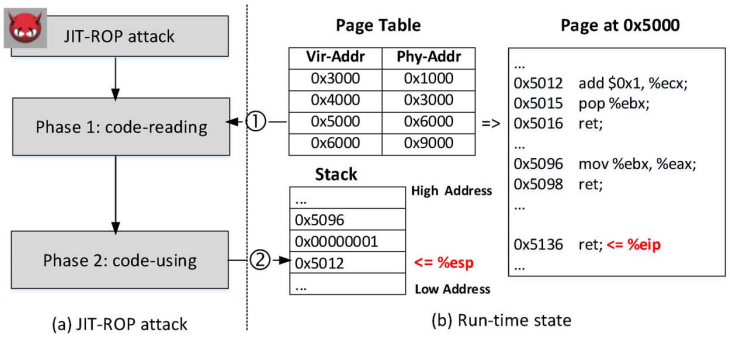

Fig. 1. A running example of JIT-ROP attacks which illustrates ① exploiting a memory disclosure and ② manipulating an ROP payload on stack.

In general, the aforementioned code and memory randomization is effective in defending code reuse attacks. However, they are not resilient to sophisticated code reuse attacks that exploit memory disclosure vulnerability to bypass detection. To address this limitation, adaptive randomization techniques have been proposed [24], [26], [42]. In particular, these techniques minimize the attack surface by performing randomization iteratively. For example, TASR [24] iteratively randomizes code addresses at run time, Isomeron [26] dynamically changes program execution paths, and RuntimeASLR [27] uses heavyweight instrumentation to enable re-randomization. While adaptive randomization can be used for the defense against sophisticated code reuse attacks like JIT-ROP, existing techniques [26], [42] typically introduce significant overhead, making them non-deployable in the real world. In addition, existing techniques [24], [42] rely on the access to source code or symbolic information. Considering that program developers may not disclose their code publicly, defense mechanisms based on source code or symbolic information are less practical. In this work, we develop a new defense mechanism that does not require source code access and, more importantly, introduces moderate performance overhead.

Memory Restriction. Memory restriction [20], [21], [22], [23] is also an effective efficient defense approach to obstructing code reuse attacks. In particular, it marks a memory space executable only. Thus, any read attempt to the memory is denied. Since the code reuse attack exploits vulnerabilities and discloses memory information, this technique can prevent memory disclosure and thus defend code reuse attacks. However, memory restriction typically requires hardware support. Considering some platforms are lack of such hardware support (e.g., mobile platform [43]), existing techniques are not sufficient for the defense against JIT-ROP. In this work, our defense mechanism is a software solution independent of any hardware support.

III. Overview

In this section, we discuss the problem scope of our research followed by the overview of CHAMELEON. In particular, we begin with the threat model. Then, we describe how CHAMELEON works at the high level.

A. Threat Model

Our research addresses JIT-ROP attacks, in which an adversary constructs an exploit based on the information leaked from memory corruption. As such, our threat model assumes that software contains memory disclosure vulnerabilities and an attacker can exploit these vulnerabilities to read/write arbitrary memory (without crashing the program). Considering modern operating systems have deployed DEP [1] and ALSR [13], our threat model also assumes that code pages cannot be set to executable and writable at the same time and position-independent code (PIC) is randomly loaded in memory upon startup. From the attacker's viewpoint, we finally assume that code layouts are randomized at a fine-grained granularity, making the registers [12] used and instruction locations within a function [11] or a basic block [14] different. The reason behind this assumption is that an adversary may be able to find code pointers (e.g., virtual table entries) in non-executable memory (e.g., stack and heap), infers the code layout of the rest of the memory without directly reading them, and finally mounts an indirect JIT-ROP attacks [20]. From the perspective of the defense side, we assume security engineers only have access to binary code because software authors may not disclose their source code. Considering binary obfuscation techniques [44], [45] (e.g., code packing and encryption) typically make a program sluggish at run time, we assume the binary code is not obfuscated. Data only attacks [46], [47], [48], [49] that modify the data objects are outside the scope of this paper.

B. CHAMELEON

To thwart JIT-ROP attacks, we design CHAMELEON, a practical defense mechanism that consists of two parts – *Page Table Transformation* and *Address Translation*. The page table transformation dynamically varies a memory layout by mapping code page addresses to a new code space that has not yet been assigned to any code or data. More specifically, the page table transformation replaces each original code page address with a new virtual address. Since CHAMELEON conceals this one-to-one transformation and does not update code pointers in binary accordingly, leaking an address at run time (or from a binary) only allows an attacker to infer the original memory

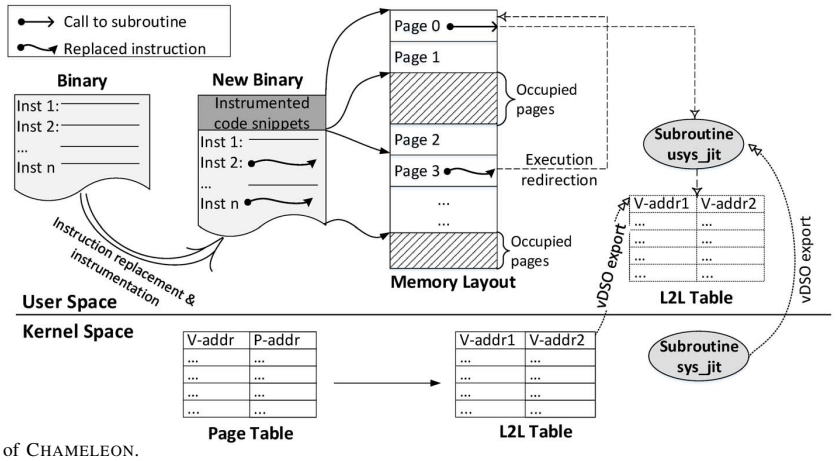

Fig. 2. The architecture of CHAMELEON.

layout. Thus, using it to dump code instructions is no longer effective.

Considering page table transformation also brings side effects to programs running in user space, CHAMELEON further uses an address translation mechanism that coordinates page table transformation and ensures the correctness of program execution. In address translation, CHAMELEON first identifies a set of instructions in a binary file. Then, it replaces these instructions with a pre-defined instruction and instruments a new code snippet to the binary. When this instrumented binary is loaded and the replaced instruction is executed, the program is automatically redirected to the aforementioned code snippet. The code snippet contains a set of instructions that take two actions to ensure the correctness of the program execution. First, it saves the program execution status. Our instrumentation changes the control flow of the program. When the program enters the instrumented code snippet for address translation, therefore, CHAMELEON needs to maintain the original program logic. Second, the code snippet coordinates with a subroutine to perform an address translation transparently. The change to the page table may incur program crash if code addresses are not updated properly. While upon program execution, as a result, CHAMELEON translates the original code addresses to their new mapping.

Figure 2 shows the architecture of CHAMELEON, indicating how we design and implement CHAMELEON to perform page table transformation and address translation. In Section IV and V, we will provide the detail about how we design and implement page table transformation and address translation in CHAMELEON.

IV. PAGE TABLE TRANSFORMATION

As is mentioned above, the page table transformation is one component in CHAMELEON. It dynamically shuffles code page addresses making code addresses non-deterministic. In this way, an attacker is not able to infer code addresses, and thus dump code instructions for constructing a malicious ROP payload. In this section, we describe the design and implementation of page table transformation in detail.

A. Design

We design the page table transformation as an address randomization mechanism. In particular, it first initiates an L2L table – a mapping table for address transformation – right after an OS loads a binary file and creates its page table. In particular, our randomization mechanism parses the ELF header of the binary file and identifies the code segment as well as its code page addresses. For a binary file that links to dynamic libraries, our randomization mechanism further identifies the code page addresses of each library. Searching these code page addresses in the page table, our randomization mechanism then replaces the address of each code page with a virtual page address that has not yet been assigned to any code or data pages. At the same time, this randomization mechanism also records this replacement in the L2L table. To prevent an attacker from exploiting overread vulnerabilities and leaking a code page, our randomization mechanism also ensures the transformed code page address is not adjacent to the addresses assigned to data pages.

In fact, transforming code page addresses to an unknown space does not provide sufficient protection. When an attacker attempts to dump a code page with its original address, the processor's MMU detects the page fault. Further, it triggers the page fault handler to relocate the missing code page in the code segment. If it succeeds, the page fault handler inserts into the page table a new entry with its virtual address equal to the address that the attacker attempts to exploit. Thus, the attacker still has the capability to dump code instructions and exploit the program. To address this problem, our randomization mechanism inserts a placeholder entry. More specifically, after overwriting the original code page address in a page table, the randomization mechanism inserts into the page table an entry with the physical address pointing to Null and virtual address equal to the original code page address. In this way, when an

attacker attempts to exploit an original address, the OS kernel identifies this address in the page table but does not disclose any code instructions.

While our address randomization mechanism increases the difficulty for an attacker in dumping code instructions, one-time randomization is not sufficient because an attacker can use a brute-force approach to traverse address space and identify valid code address. In this design, we address this problem by extending our randomization mechanism with an iterative address shuffling mechanism. In particular, our randomization mechanism performs address re-transformation on the basis of a pre-defined interval or every time an I/O system call is invoked. In address re-transformation, our randomization mechanism shuffles virtual addresses of code pages in the page table.

To coordinate the programs running in user space, last but not least, we also design an interface to facilitate address translation. With this interface, a program in user space can retrieve a transformed address by querying the L2L table with an original code page address, and vice versa.

B. Implementation

We implemented the page table transformation inside the Linux kernel. Here we introduce and discuss some important implementation details.

In general, a Linux operating system does not load an entire binary file into memory due to performance optimization. At run time, it swaps in and out the code pages of a program frequently, and makes its page table dynamically varying. This dynamics increases the difficulty in our implementation, and even results in the malfunction of the aforementioned randomization mechanism. To simplify our implementation and ensure the effectiveness of our randomization mechanism, our implementation uses memory lock function `mlock` that forces a Linux operating system to load all the code pages of a program into memory and disables it to swap them out. As we will show in Section VI, such an implementation does not incur a significantly high memory usage for the simple reason that the code pages of a program usually consume less memory space.

As is mentioned above, the page table transformation replaces code page addresses with virtual addresses that have not been assigned to any code or data pages. Considering a Linux operating system can also allocate these unused virtual addresses at run time, our implementation marks the virtual addresses used for address transformation and denies the allocation requests to them. In addition, our implementation duplicates the L2L table every time a process forks a child process.

To deal with address space shuffling, our implementation also creates a new kernel thread – `kth_l2l`. Since CPU memory management unit uses the Translation Lookaside Buffer (TLB) to speedup the mapping from a virtual memory page to physical memory, and our address shuffling incurs the data inconsistency between TLB and a page table, the implementation of `kth_l2l` thread is for the purpose of refreshing TLB every time address re-transformation is performed.

In addition, `kth_l2l` thread is responsible for coordinating program execution. When address re-transformation is performed, a program might be in an execution status. Without taking an appropriate action in a Linux operating system, the program execution might crash. In our implementation, we use `kth_l2l` to pause program execution when address re-transformation is performed. More specifically, `kth_l2l` thread triggers inter-processor interrupts and moves running threads of the program to a waiting queue. After address re-transformation completes, `kth_l2l` thread updates the program counter of the threads in the waiting queue, signals all CPU cores and moves the paused threads back to the running queue. Note that our implementation also includes a watchdog mechanism. With this mechanism, thread `kth_l2l` delays address re-transformation if address translation is in process.

V. Address Translation

The aforementioned page table transformation shuffles code page addresses for preventing attackers from dumping code instructions. However, this randomization mechanism also throws the binary code into disarray and even introduces program malfunction at run time. We address this problem by designing and implementing an address translation mechanism. In this section, we describe our design and implementation in detail.

A. Design

At the high level, our address translation mechanism first uses a disassembler to recover the semantics of a binary file. Guided by the disassembly, it then modifies the binary file by performing machine code replacement and instrumentation. Finally, our address translation mechanism loads this new binary file in memory and coordinates with a subroutine to complete address translation. Here, we describe important design details as follows.

Disassembler. To recover the semantics of a binary file, we design a disassembler by extending a popular open source disassembler [50] with the state-of-the-art technique described in [35]. Considering the complexity of binary code and the limitation of our extended disassembler, in our design, we design the disassembler in a way that it ignores the binary code with uncertainty. By uncertainty, we mean the code that cannot be statically disassembled. For example, when we have insufficient information to statically determine a function. In other words, when our disassembler is not confident in the correctness of the disassembly, it automatically ignores that code snippet and moves to the next code snippet in that binary file. As we will show and discuss later in this paper, this design does not impact the effectiveness in mitigating JIT-ROP attacks.

Code replacement and instrumentation. We design a binary instrumentation mechanism that uses disassembly information to perform instruction replacement and instrumentation. When a CPU executes an instruction, and the instruction does not perform a cross-page visit, a program counter moves within a

page. In this case, the transformed page table does not destroy program execution. As a result, our binary instrumentation mechanism does not replace instructions or instrument the binary file in such type.

In fact, there are only a few instructions that might incur cross-page visits, and these instructions include direct `call`, direct `jmp` and the last instruction in a code page. In our design, the binary instrumentation mechanism uses disassembly information to identify their positions in the binary and modifies the binary by replacing these instructions with an instruction pair – `push` and `ret`. Considering indirect jump instructions (e.g., `ret`) might also incur cross-page visits, and we cannot use disassembly information to determine if they incur cross-page visits, we replace all indirect jumps with the instruction pair.

The aforementioned address transformation influences program logic. In this design, the aforementioned instruction pair redirects program execution to an individual code snippet responsible for maintaining the integrity of program logic. Along with each instruction replacement, our binary instrumentation mechanism therefore instruments the binary file with a code snippet. In particular, we design the code snippet to interact with a subroutine that translates a code address used in the binary to the transformed address. Using this code snippet, a program is able to follow its original program logic.

B. Implementation

Guided by the aforementioned design, we implemented address translation on Linux operating system as follows.

Disassembler. By extending an open-source disassembler – `Dyninst` [50], we implemented a disassembler that improves the disassembling capability of `Dyninst`. Guided by the state-of-the-art techniques in [35], in particular, we augmented `Dyninst` with the ability of identifying exception handlers, exported symbols, and return addresses. In addition, the extended version of `Dyninst` includes a new linear algorithm introduced in [35]. Along with other source code developed in this work, we will release our disassembler and make it publicly available on Github.

Instruction replacement. For instruction replacement, our implementation first uses disassembly information to identify the binary code representing the instructions that change program execution from one page to another. Then, our implementation replaces each cross-page instruction with instruction `push` and `ret`. In `X86` architecture, instructions have varying length. A simple instruction replacement without considering the length variation may destroy the offset between instructions, and incur unexpected program corruption. In our implementation, we address this problem as follows.

For a cross-page instruction that shares the same length with the instruction `push` and `ret`, we make instruction replacement without taking further actions. For the instruction that takes more space in memory, we swap it with the instruction pair and pad the unoccupied space with `nop` instructions.

For a cross-page instruction short in length, our replacement strategy groups the adjacent instructions in front and swaps the group with instruction `push` and `ret`. Then, we reallocate the adjacent instructions in a separate code snippet – *a trampoline*. Figure 3 illustrates an example of instruction replacement. As is shown in the figure, we identify a cross-page instruction represented by a 5-byte `call` instruction. To replace it with instruction `[push $0x8002096]` and `ret`, we group instruction `[call 0x804d990]` and `[shl $0x3,%eax]`, and perform instruction swapping and padding accordingly. Then, we reallocate swapped instruction `[shl $0x3,%eax]` in a trampoline.

Binary instrumentation. In our implementation, we instrument the binary file with the aforementioned trampolines. In addition to migrated instructions, a trampoline contains another two sets of instructions. As is discussed above, the trampoline is designed for maintaining the integrity of program logic. As a result, one set of instructions is used for saving program execution status before original program logic is redirected, while the other is for calling a subroutine to complete address translation. Figure 3 shows an example to illustrate our implementation. As is shown in the figure, the original program logic is to execute instruction `[call 0x804d990]` and then return to address `0x804ae69`. In this example, we replace instruction `[call 0x804d990]`. To ensure the program returns back to its original logic, our implementation therefore uses instruction `[push $0x804ae69]` to maintain the return address on stack. In addition, our implementation saves register status using instruction `pushf` and `pusha`. Following these instructions, three instructions are included to invoke a subroutine `usys_jit`. In particular, the first two instructions are used for setting two arguments for `usys_jit` with the values equal to `0x804d990` and `0x2`. Here, `0x804d990` represents the target address of the replaced instruction while `0x02` specifies instruction `push` and `ret` substitute instruction `call`. Note that, the trampoline also contains an effective instruction `lea` that allocates a 4-byte space on stack, which the subroutine uses to store the transformed address. Since our implementation replaces instructions with instruction pairs that invoke a trampoline located at a specified address, we need to determine the specific address of each trampoline in advance of the binary instrumentation. To do it, we make all trampoline addresses deterministic in memory.

More specifically, our implementation first identifies the dynamic libraries that a program links to. As a dynamic library is typically loaded at a random address in memory, and instruction pair `push` and `ret` needs a specific address, randomly allocating the trampolines tied to the dynamic library in memory may incur execution failures. Our implementation addresses this problem by loading them at pre-determined individual code pages. Thus, their addresses can be known in advance. Note that 32-bit Linux operating system loads a program starting at `0x8048000`, and our implementation always loads the trampolines tied to dynamic libraries at the memory space between `0x7000000` and `0x8048000`.

For those trampolines tied to a program, our implementation simply places them ahead of the code segment of the binary file because the loading address of the program is known in advance. To ensure trampolines can be loaded correctly, our

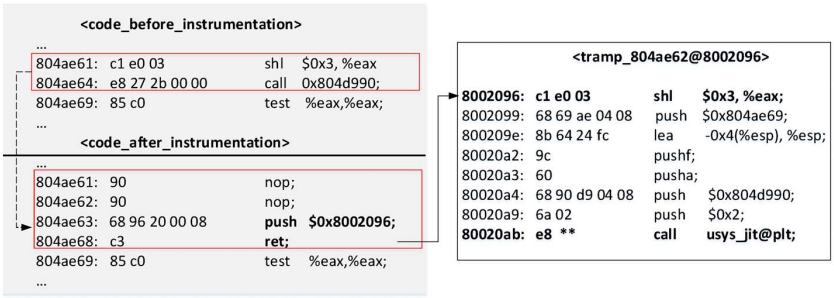

Fig. 3. An example of code replacement and instrumentation.

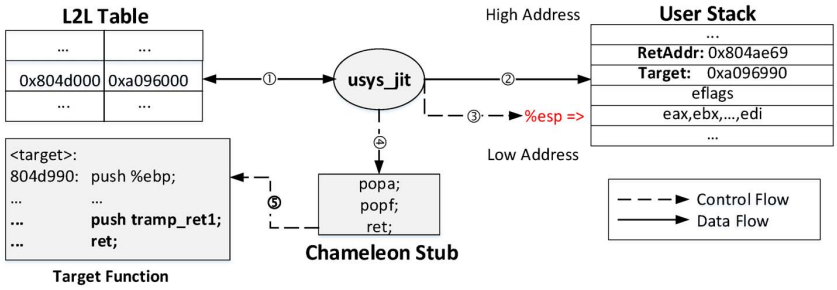

Fig. 4. The work flow of `usys_jit`.

implementation updates the ELF header of the binary file and thus adjusts the starting address of the code segment. Considering the aforementioned address randomization destroys the correctness of program execution if applied to newly added trampolines, our implementation ensures the trampolines and the original code segment are loaded in different code pages.

With all trampolines in their own code pages, our implementation can guarantee the correctness of program execution by not churning their code page addresses. While this implementation places these code pages off the protection of address randomization, it does not jeopardize security. As is illustrated in Figure 3, a trampoline typically does not contain gadgets necessary for constructing malicious payloads. Even though address randomization is not applied to them, therefore, an attacker cannot exploit trampolines to launch attacks. Considering an attacker may search unintended gadgets in the trampoline and launch attacks, we further implemented a control flow integrity check mechanism to ensure one cannot jump in and out a trampoline without executing all the instructions defined in the trampoline. As is presented above, our instrumentation moves all indirect jump into the trampolines. We, therefore, enforce the check in the trampolines to guarantee the above requirement of control flow integrity.

Subroutine. As is shown in Figure 3, a trampoline invokes subroutine `usys_jit` to complete address translation. In particular, `usys_jit` examines the instruction replaced by `push` and `ret`. For an instruction that does not transfer control flow (e.g., conditional jump instructions, the conditions of which do not hold), `usys_jit` does not perform address translation. Rather, it invokes a stub that restores program execution status and redirects program execution to its original logic. For an instruction that transfers program control flow from one page to another, before calling the stub, `usys_jit` performs a lookup in the aforementioned L2L table, retrieves the transformed address and eventually stores it at the reserved space on stack. To illustrate this, Figure 4 shows a running example.

As is shown in the example in Figure 4, when `usys_jit` is called, it first extracts the argument stored on stack and determines the type of the instruction replaced by the pair of push and return. In this example, the replaced instruction is `call`, which certainly transfers the program control flow to another code page. Therefore, `usys_jit` further performs a query to the L2L table with another argument that indicates the original code address `0x804d990` (①), and stores the query result `0xa096990` on stack (②). Before exiting, `usys_jit` sets stack pointer to the space where the original register states are stored (③), and then redirects execution to the stub. The stub restores the original states of the general register and status register (④). After this, the stack pointer links to the space where `usys_jit` stores transformed code address (④). When the stub returns, therefore, the program is redirected and follows its correct logic (⑤).

In our implementation, subroutine `usys_jit` is a kernel space routine exported to user space programs. As calling a kernel routine through a system call interface introduces a context switch, which incurs significant performance overhead, our implementation minimizes system overhead by using virtual Dynamic Shared Object (vDSO), a Linux kernel mech-

anism [51]. In particular, we first implemented system call `sys_jit` that interacts with the aforementioned L2L table and performs address translation. Then, we utilized vDSO to export the L2L table and `sys_jit` to user space programs so that programs can call `sys_jit` in user space (i.e., invoking `usys_jit`) and perform address translation in-process without incurring the performance penalty of a context switch.

While this implementation places security sensitive processes and data into insecure user space, it does not compromise our security guarantee. By design, Linux operating system loads vDSO and vVar at a random location in memory. Thus, an attacker cannot obtain its address in advance. This means that, it is difficult for an attacker to pre-know the address of `usys_jit` and L2L. However, the address of `usys_jit` is hard-coded in the trampolines that are placed at fixed locations. An adversary may firstly infer the locations of our trampolines and then the location of `usys_jit`. Further, she could read `usys_jit` out to learn address of the L2L table and then contents in the L2L table. Although such an attack against the L2L table is possible, our re-randomization of the L2L table will make any leaked contents obsolete quickly or even immediately after the leaking.

C. Discussion

Here, we discuss some implementation details that are not mentioned above but worth discussion.

To ensure the correctness of program execution, as is discussed above, our extended disassembler automatically ignores the code snippets with uncertainty. CHAMELEON does not shuffle the addresses of the code pages containing these code snippets nor perform binary instrumentation, making these code pages stationary. However, this design is not sufficient. When a program control flow switches from a stationary code page to a shuffled code page (i.e., a code page with its address randomized), the program cannot be correctly redirected to the corresponding code page because of the address transformation. In our implementation, we augment our design and implementation as follows.

In advance of entering a stationary code page, our implementation sets all the shuffled code pages non-executable. In this way, the operating system throws an error when the program attempts to switch from the stationary page to a shuffled page. To catch this error, we implement a kernel thread – `kth_jit`. Since `kth_jit` resides in kernel and is granted the access to page table and L2L table, our implementation leverages this ability to perform address translation, set all shuffled code pages executable and eventually redirect program execution to the correct code page.

In Linux, a program may use system call such as `sigaction` to register a function that responds a specific signal. On receipt of a specific signal, in particular, the operating system pauses program execution, retrieves the address of the registered function and executes the instructions defined in the registered function at user space. After execution completion, the program invokes system call `sigreturn` that switches execution context back to the kernel and resumes program execution. To ensure address transformation does not destroy signal handling, our implementation also hooks the kernel function `setup_frame` so that it can coordinate with L2L table and perform address translation.

D. Results

Disassembly. Table I shows the complexities of the programs (i.e., lines of instructions) in our test cases, as well as the capability of CHAMELEON in disassembling each program. We observe our disassembler can correctly disassemble the programs in our test cases without ignoring any machine code. This indicates that our disassembler has the ability of handling programs with different complexities. As this ability implies CHAMELEON can apply randomization to any code page addresses of a program, the results in Table I further indicate CHAMELEON can provide programs with sufficient protection. We also evaluate how many pages for the trampoline that remain stationary, there are 18.3% trampoline pages on average for all of our tested programs.

Memory consumption. As is mentioned above, we design two settings to study memory usage before and after binary instrumentation. The last two columns in Table I show the results of our experiment in the first setting. We observe that the number of code pages that need to be loaded in memory increases by 25%∼86% after binary instrumentation. Although this increase in memory usage is significant, its influence upon memory consumption is negligible. When a binary file is loaded and run, code pages only carry little weight to memory consumption. In Linux operating system, the size of a page is typically 4KB. As shown in Table I, even for the program with a large amount of instructions, the memory overhead imposed by binary instrumentation is minimal (e.g., 1.8MB for `gcc`).

Figure 5 shows the results of our experiment in the second setting. As is shown in the figure, the memory overhead is negligible in all test cases with an average overhead of 1.38%. Again, this indicates that instructions typically carry little weight to memory consumption and the solution of our binary instrumentation in CHAMELEON is not intrusive.

VI. EVALUATION

In this section, we evaluate our CHAMELEON prototype both empirically and theoretically. Empirically, in particular, we test CHAMELEON on a variety of real-world software and thus evaluate its correctness and intrusiveness. Theoretically, we analyze the effectiveness of CHAMELEON in mitigating JIT-ROP attacks.

A. Experimental Design

To evaluate CHAMELEON in general, we selected 16 representative test cases, including three web service applications (`nginx 1.6.0`, `httpd 2.2.27` and `lighthttpd 1.4.35`), one compression program `xz 5.1.4` and twelve benchmark programs from SPEC CPU2006. Along with these test cases, our test case set also includes 12 dynamic libraries that these representative programs link to. We listed all these test cases in Table Ia and Ib. We ran all these test cases

Program	# of instructions (K)		# of code snippet ignored		# of code pages	
	-O2	-O3	-O2	-O3	Original	New
httpd	111	102	0	0	94	146
nginx	163	128	0	0	133	179
lighthttpd	40	36	0	0	34	52
xz	12	14	0	0	11	18
gcc	685	791	0	0	612	1073
bzip2	13	15	0	0	12	15
gobmk	180	210	0	0	160	246
h264ref	111	195	0	0	106	137
hmmer	64	71	0	0	54	84
perlbench	235	270	0	0	207	274
sjeng	24	28	0	0	25	36
astar	9	11	0	0	8	11
omnetpp	124	138	0	0	107	199
libquantum	10	11	0	0	8	12
specrand	0.3	0.3	0	0	1	1
mcf	3	3	0	0	3	4

(a) Programs in 16 test cases.

Library	# of instructions (K)		# of code snippet ignored		# of code pages	
	-O2	-O3	-O2	-O3	Original	New
ld-linux	28	31	0	0	26	39
libapr-1	39	42	0	0	31	55
libaprutil-1	26	31	0	0	23	36
libcrypt	6	7	0	0	6	9
libc	347	381	0	0	303	473
libdb-4	336	345	0	0	307	477
libdl	1	1	0	0	2	3
libexpat	29	36	0	0	26	33
liblzma	31	36	0	0	28	40
libm	69	70	0	0	53	75
libpcre2-8	49	51	0	0	45	75
libpthread	16	17	0	0	14	24

(b) Dynamic libraries that test cases link to.

TABLE I. THE STATISTICS OF BINARY INSTRUMENTATION. NOTE THAT -O2 AND -O3 INDICATE THE OPTIMIZATION OPTION OF gcc COMPILER.

Program	httpd	nginx	light-httpd	xz	gcc	bzip2	gobmk	h264ref
	1,400	4,205	2,129	2	5,792	46,996	2,754	3,949
Program	hmmer	perlbench	sjeng	astar	omnetpp	libquantum	specrand	mcf
	1,010	2,500	3,099	4,278	7,830	20	0.1	897

TABLE II. THE FREQUENCIES AT WHICH A PROGRAM NEEDS TO INTERACT WITH A TRAMPOLINE. NOTE THAT THE FIGURE IS AT THE SCALE OF 10^3.

on a desktop equipped with Intel(R) Core2 Duo CPU E8400 3.00GHz and 2GB RAM.

To evaluate CHAMELEON from different perspectives, we configured the aforementioned test cases accordingly. As is discussed in Section V, our disassembler automatically ignores the code snippet with uncertainty. In CHAMELEON, we do not obfuscate the address of the page containing that code snippet for the simple reason that the ignored code snippet may contain cross-page instructions and shuffling such a page without instruction replacement incurs program corruption. If there are too many such code snippets left without concealing, therefore, CHAMELEON cannot provide sufficient security guarantee. Considering this, we design an experiment to examine the number of code snippets that our disassembler ignores. In particular, we used `gcc 4.8` to compile the programs used in our test cases, and then disassembled them with our disassembler. In compiling a program, we enabled the optimization option of `gcc` – i.e., setting parameter `-O2` and `-O3`, respectively. In addition, we enabled the debug option of `gcc`, allowing us to count the code snippets ignored by our disassembler.

In CHAMELEON, we replace machine code and instrument binary files. In addition, we load all code pages to memory. Both instrumented code and the program loading mechanism increase memory usage. As a result, we further design an experiment to examine memory consumption. In particular, we compare the memory usage of CHAMELEON with two different baselines. Our first baseline is the memory usage in the setting, where we force operating system to load all the code pages in memory when loading a program without instrumentation. By default, operating system swaps in and out code pages, and memory consumption varies dynamically. Therefore, our second baseline represents the memory consumption in the setting where we allow the operating system to manage code pages in an autonomous manner.

Considering that page table randomization and address translation introduce additional operations and thus disturb programs at run time, last but not least, we design an experiment to further study the intrusiveness of CHAMELEON. In particular, we take standard SPEC CPU2006 benchmark and observe the performance overhead of a program in different randomization strategies, i.e., instant code page randomization at different frequencies as well as randomization at each system call invoked. To obtain an accurate measurement on performance overhead, we ran web service applications by using ApachBench with command `ab -k -n50000 -c100 HOST/index.html`,

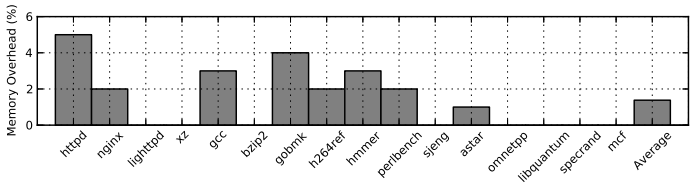

Fig. 5. Memory overhead of CHAMELEON.

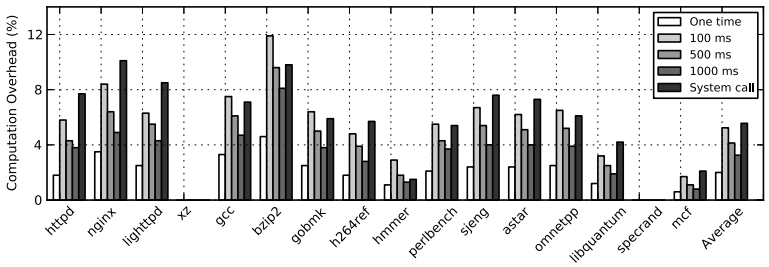

Fig. 6. Computation overhead of CHAMELEON in different randomization strategies.

and compression program `xz` by performing compression on a 16MB binary file for 1,000 times. For the rest of the programs mentioned above, we ran them using standard SPEC CPU2006 benchmark.

Computation. Figure 6 illustrates the computation overhead that CHAMELEON imposes to programs. In specific, we observe the computation overheads when CHAMELEON performs randomization only once or performs re-randomization at the following frequencies: every $100ms$, every $500ms$, every $1000ms$, and every I/O system call.

For program `xz` and `specrand` shown in Figure 6, we also observe that CHAMELEON imposes extremely low computation overhead even though performing randomization highly frequently. As is shown in Table II, we also discover that both programs are rarely redirected to a trampoline in comparison with other programs. Since address translation takes places in a trampoline, this interesting observation implies that address randomization exhibits less intrusive to a program when the program has less address translation requests. The reason behind this surprising observation is as follows.

At the moment a program is redirected to a trampoline and enters address translation, any variation in a page table may incur execution failures. As such, CHAMELEON delays address randomization if a program is in the process of address translation. When a program performs less address translation, the delay of address randomization is significantly reduced. Thus, we observe that computation overhead imposed to `xz` and `specrand` is close to zero.

As we mentioned in Section IV, `kth_l2l` pauses the program execution when address re-transformation is performed, the address re-transformation takes 2.39ms on average.

B. Theoretical Analysis

As is mentioned in Section IV, an attacker might attempt to dump a code page by inferring its address in a brute-force manner. In other words, he might attempt to exploit all possible code page addresses until a valid code page address is found. In this work, CHAMELEON counteracts such a practice using an instant randomization mechanism. Here, we theoretically analyze the effectiveness of this defense mechanism.

More specifically, we model our problem as follows. Suppose a memory space is divided into N pages, and an operating system needs to randomly allocate y slots in N pages for program p. In this setting, we also assume an attacker attempts to dump x code pages of program p ($x \leq y$), and $f(x)$ is the number of attempts needed for an attacker to identify a valid code page.

Since an attacker does not have knowledge about the allocation of the program in memory, the probability for him to correctly identify one valid code page is $\frac{y}{N}$. In other words, an attacker needs to make $\frac{N}{y}$ attempts in order to dump one code page of program p. Given the code pages that an attacker has already dumped, the probability of identifying the next code page decreases accordingly. For example, an attacker dumps $x-1$ code pages and the probability for him to identify the next valid one is $\frac{y-x}{N-x}$. To dump x code pages of program p, an attacker needs at least $x \cdot \frac{N}{y}$ attempts, i.e., $f(x) = \frac{N}{y} + \frac{N-1}{y-1} + ... + \frac{N-(x-1)}{y-(x-1)} \geq x \cdot \frac{N}{y}$.

Using the theoretical model above, we further conduct our analysis with a real-world setting. In particular, we consider a 64-bit version of Linux operating system with 50% memory space allocated to user space. In such a system, the page size is typically 4KB, and our empirical observation indicates Linux operating system generally needs to allocate about 500 code pages for a program and libraries the program links. To identify one of these code pages in memory, theoretically speaking, an attacker therefore needs to make at least 2^{42} attempts, i.e., $f(x) \geq x \cdot \frac{\frac{r \cdot 2^n}{s}}{y} = \frac{x \cdot r \cdot 2^n}{y \cdot s}$ where $x = 1$.

As is discussed in Section IV, CHAMELEON can perform address space randomization every time an I/O system call is invoked. In practice, it is impossible for an attacker to make 2^{42} attempts without triggering re-randomization because each

exploit attempt incurs an I/O operation.

VII. Discussions

In this section, we discuss the security effectiveness provided by our defense under different types of attacks.

ROP without memory disclosure. Without memory disclosure, the attacker can only infer the code layout at run-time from code layout in static binaries. However, our defense is compatible with ALSR and more importantly, we further introduce fine-grained code layout randomization. Such randomization mechanisms make the aforementioned inferring extremely difficult or impossible.

JIT-ROP. To perform JIT-ROP attacks, an adversary must read code pages from the memory using their virtual addresses. Our argument against JIT-ROP is twofold. First, as described in Section IV, CHAMELEON transforms code space to an unknown space and inserts placeholder entry to prevent reading the code. For example, to prevent HeartBleed attack [52] reading beyond the data object, the transformed code space is not adjacent to the data, and the placeholder prevents disclosing any code. Second, our defense frequently re-randomizes virtual addresses of code pages, which essentially prevents the adversary from knowing those addresses in advance. Therefore, the adversary can only opt to brute force the whole address space, which will certainly trigger paging errors and raise alert to the security administrators.

Our defense is compatible with code generated by just-in-time compilation, such as code compiled from `javascript`. To protect such code, we just need to instrument the just-in-time compiler so that our instrumentation is enforced in the generated code. To secure L2L table, either obfuscating L2L table access by using ambiguous instructions or moving the L2L table at runtime are two potential methods.

CHAMELEON made a couple of modifications on OS kernel. However, it does not mean CHAMELEON is not a practical solution. CHAMELEON is compatible with legacy programs, even though a program does not need protection, it can still run on the OS modified by CHAMELEON.

Indirect JIT-ROP. As is explained in our threat model, we require the code layouts to be randomized at a fine-grained granularity, making the registers used and instruction locations within a function or basic block different. This static randomization prevent attackers from inferring the code layout merely based on code pointers. Therefore, it is very difficult to mount indirect JIT-ROP attacks.

Other attacks. Our defense system currently cannot prevent ret2libc attacks, since we do not prevent all possible control flow hijacking and allow any control flow to the entry points of functions. In fact, preventing ret2libc attacks is a still an open problem in the research community.

VIII. Conclusion

The practicality of existing address space randomization techniques has been shown to be insufficient for defending sophisticated JIT-ROP attacks. In this paper, we present CHAMELEON, a new practical defense mechanism. It leverages an instant effective address space randomization mechanism to obstruct code page disclosure and thus take over the initiative in the attacking-vs-defending war. Assisted by state-of-the-art disassembly techniques, we implement CHAMELEON in an elegant and effective way. Experiments with over 16 real-world programs enable us to announce that (1) CHAMELEON can fundamentally prohibit attackers from intruding the protected programs and (2) CHAMELEON only imposes moderate performance overhead. While we implement CHAMELEON on a 32-bit version of Linux operating system, our design principle is general. As part of the future work, we therefore will extend the implementation of CHAMELEON to a 64-bit version of Linux operating system. Considering our design is also hardware independent, we will also explore the implementation of CHAMELEON over other computing platforms, like mobile devices.

IX. Acknowledgement

We would like to thank Jun Wang and Yu Hou for their valuable technical supports, and professor Zhiqiang Lin for his valuable comments on this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, NSF CNS-1505664, and Chinese National Natural Science Foundation (NSFC 61073027, NSFC 61272078, NSFC 61321491).

References

[1] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2." 2008, http://support.microsoft.com/kb/875352.

[2] I. Molnar, "Exec Shield," 2003, http://people.redhat.com/mingo/exec-shield/.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*, 2011.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS '10)*, 2010.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *USENIX Conference on Security (Security '11)*, 2011.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[7] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security Symposium (Security '14)*, 2014.

[8] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *USENIX Security Symposium (Security '03)*, 2003.

[9] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX Security Symposium (Security '05)*, 2005.

[10] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "Ilr: Where'd my gadgets go?" in *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.

[11] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference (ACSAC '06)*, 2006.

[12] V. Pappas, M. Polychronakis, and A. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy (Oakland '12)*, 2012.

[13] P. Team, "Pax address space layout randomization (aslr)," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[14] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[15] J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent runtime randomization for security," in *International Symposium on Reliable Distributed Systems (SRDS '03)*, 2003.

[16] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, 2016, pp. 467–478.

[17] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[18] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '10)*, 2011.

[19] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Second European Workshop on System Security*, 2009.

[20] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Symposium on Security and Privacy (Oakland '15)*, 2015.

[21] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*, 2015.

[22] B. Kjell, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[23] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[24] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd Conference on Computer and Communications Security (CCS '15)*, 2015.

[25] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY '16)*, 2016.

[26] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[27] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make aslr win the clone wars: Runtime re-randomization," in *Network and Distributed System Security Symposium (NDSS '16)*, 2016.

[28] S. Designer, ""return-to-libc" attack," *Bugtraq*, August 1997.

[29] Nergal, "The advanced return-into-lib(c) exploits," *Phrack Magazine*, vol. 58, no. 4, 2001.

[30] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS '05)*, 2005.

[31] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Annual Computer Security Applications Conference (ACSAC '11)*, 2011.

[32] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones," in *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[33] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy (Oakland '10)*, 2010.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[35] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *USENIX Conference on Security (Security '13)*, 2013.

[36] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *IEEE Symposium on Security and Privacy (Oakland '14)*, 2014.

[37] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *IEEE Symposium on Security and Privacy (Oakland '08)*, 2008.

[38] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '15)*, 2015, pp. 144–164.

[39] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *USENIX Security Symposium (Security '14)*, 2014.

[40] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control- flow integrity protection," in *USENIX Security Symposium (Security '14)*, 2014.

[41] C. Nicholas, B. Antonio, P. Mathias, W. David, and R. G. Thomas, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security Symposium (Security'15)*, 2015.

[42] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Conference on Security Symposium (Security '12)*, 2012.

[43] Intel, "Intel atom processor for smartphone and tablet," http://ark.intel.com/search/advanced?s=t&FamilyText=Intel

[44] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, 2013.

[45] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security (CCS'03)*, 2003.

[46] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *European Symposium on Research in Computer Security (ESORICS '15)*, 2015.

[47] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th Conference on USENIX Security Symposium (Security '05)*, 2005.

[48] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium (Security '15)*, 2015.

[49] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland '16)*, 2016.

[50] Dyninst, "Dyninst programmer's guide," 2013, www.dyninst.org/sites/default/files/manuals/dyninst/DyninstAPI.pdf.

[51] J. Corbet, "On vsyscalls and the vdso," http://lwn.net/Articles/446528/, 2011.

[52] T. Lee, "Heartbleed (cve-2014-0160) test & exploit python script," 2014, https://gist.github.com/eelsivart/10174134.