This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

Stopping Memory Disclosures via Diversification and Replicated Execution

Kangjie Lu*, Meng Xu†, Chengyu Song‡, Taesoo Kim†, Wenke Lee†
*University of Minnesota, †Georgia Institute of Technology, ‡University of California, Riverside

Abstract—With the wide deployment of security mechanisms such as Address Space Layout Randomization (ASLR), memory disclosures have become a prerequisite for critical memory-corruption attacks (e.g., code-reuse attack)—adversaries are forced to exploit memory disclosures to circumvent ASLR as the first step. As a result, the security threats of memory disclosures are now significantly aggravated—they break not only data confidentiality but also the effectiveness of security mechanisms. In this paper, we propose a general detection methodology and develop a system to stop memory disclosures. We observe that memory disclosures are not root causes but rather consequences of a variety of hard-to-detect program errors such as memory corruption and uninitialized read. We thus propose a replicated execution–based methodology to generally detect memory disclosures, regardless of their causes. We realize this methodology with BUDDY: By seamlessly maintaining two identical running instances of a target program and diversifying only its target data, BUDDY can accurately detects memory disclosures of the data, as doing so will result in the two instances outputting different values. Extensive evaluation results show that BUDDY is reliable and efficient while stopping real memory disclosures such as the Heartbleed leak.

Index Terms—Memory disclosure, diversification, replicated execution, N-version system, code-reuse attack

1 INTRODUCTION

MODERN systems widely deploy randomization-based security mechanisms such as Address Space Layout Randomization (ASLR) to prevent a variety of attacks (e.g., code-reuse and privilege escalation attacks). These attacks generally require overwriting a code or data pointer with an address of unintended code or data pieces. Doing so requires attackers to know both the address of the pointer and the address of unintended code or data pieces, whereby both of them are randomized by ASLR. ASLR promises to stop such attacks by making these addresses unpredictable. The effectiveness of ASLR however completely relies on the confidentiality of the randomized addresses which are ubiquitous in memory. In practice, ASLR is weak because of memory disclosures. Attackers have commonly exploited memory disclosures to reveal the randomized address to circumvent ASLR and then to launch the subsequent attacks. For example, recently proposed advanced code reuse attacks [1], [2], [3], [4] all disclose a randomized code or data pointer to bypass ASLR and then launch code-reuse attacks. In addition, memory disclosures directly result in the loss of sensitive data such as private keys. For example, the HeartBleed [5] vulnerability allowed attackers to remotely read protected memory from an estimated 24-55% of popular HTTPS sites. More critically, according to vulnerability databases [6], the number of memory disclosures is increasing in general. As a result, memory disclosures are posing a critical threat to our systems.

A memory disclosure occurs when two events take place sequentially: (1) an unintended memory read (e.g., uninitialized memory read and out-of-bound read) and (2) a disclosure of the read data to the external world. The first event is the problematic operation which reads some data that is not supposed to be read, and the second event is the condition to form a memory disclosure. Since unintended reads can be caused by various program errors such as memory corruption, uninitialized read, missing check, race condition, etc., a memory disclosure is not the root cause but rather a consequence of these problems.

Recent research has attempted to detect and defend against memory disclosures of specific data or caused by specific program errors. In particular, Readactor [7], ASLR-Guard [8], and TASR [9] transform or track code pointers to prevent them from being leaked. SeCage [10] protects private keys by storing them in an isolated memory region. Memory safety techniques (e.g., SoftBound [11], CETS [12], MemorySanitizer [13] and AddressSanitizer [14]) provide a strong protection by detecting and preventing memory errors which can cause memory disclosures. Unfortunately, these protection techniques all suffer from several shortcomings. Root causes such as integer overflow, buffer overflow, and race condition are hard to detect in complex system software. While dynamic detection suffers from coverage and efficiency issues, static detection is not precise because of problems like aliasing. Runtime defense mechanisms against these problems however impose significant performance overhead. Data-protecting techniques instead protect only a very limited set of data. For example, the aforementioned protections focus on code pointers or secret key only but leave other critical data like data pointers unprotected. More importantly, prior techniques mostly employ code analysis and instrumentation techniques, which is an inherent limitation in handling dynamic code (i.e., code generated by just-in-time compilation) which is commonly used in system software such as Berkeley Packet Filter in OS kernels and the Google V8 JavaScript engine. In general, preventing memory disclosures is very challenging. End-to-end data protection techniques (e.g., code-pointer protection) suffer from false negative and extensibility, while program-error prevention

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

techniques (e.g., memory safety techniques) suffer from high performance overhead, generality, and deployment issues.

In this paper, we propose a general detection methodology and develop a new system to stop memory disclosures in a principled and practical manner. Since memory disclosures are consequences of a number of program errors, the most intuitive way to detect memory disclosures is to check if the outgoing data contains any target data (i.e., to-be-protected data) such as randomized addresses. However, such a check is hard because the forms of target data may change along the process execution, and thus we cannot tell if a value is being disclosed or not. Instead of solving this problem directly, we transform memory disclosure detection into a general equivalence-check problem. The rationale behind the idea is that by seamlessly maintaining two running instances and diversifying only the target data, we can always detect any disclosure of such data because the disclosure will always result in the two instances outputting different values. This way, we are able to capture any memory disclosure without the need of tracking the propagation of the target data or analyzing complicated code. To realize this methodology, we develop a replicated execution–based system, namely BUDDY. In BUDDY, a target process is replicated into two buddy instances which 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 such as randomized addresses. 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 memory disclosures only when the disclosed data actually leaves OS. This way, the false positive issues caused by data structure padding [15] are eliminated. With this design, in principle, BUDDY can completely (i.e., no false negatives) and accurately (i.e., no false positive) detect any memory disclosure without the need to identify or track the data in memory, which can be error-prone and expensive. Although BUDDY focuses on detecting memory disclosures, it is efficient enough to be used as a runtime prevention tool.

The diversification in BUDDY is a key challenge, which must preserve semantics and trigger divergences upon memory disclosures. To develop effective diversification schemes, we first develop a formal model and define two properties that a diversification scheme should satisfy. As examples, we then develop two diversification schemes on top of BUDDY to comprehensively detect memory disclosures caused by various memory errors. Specifically, we first use 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. This scheme is similar to the addresss space partitioning scheme [16] but improved with self-randomization that mitigates asymmetrical attacks as discussed in §7. With this scheme, any absolute address-based over-read will always 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 which 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. BUDDY is a general platform, and more diversification schemes can be applied to detect memory disclosures. For example, to detect memory disclosures caused by temporal memory errors (e.g., use-after-free), we can apply the diversification scheme from Diehard [17] which allocates objects in a random address so that an uninitialized value or a defined value after free will likely differ in buddy instances; disclosing 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.

BUDDY-based data leak detection has multiple important advantages. First, it is general. That is, no matter what the root cause is, BUDDY detects memory disclosures when they actually occur. Second, the error-prone and expensive data tracking are completely avoided. That is, BUDDY detects memory disclosures at the point when the diversified data is actually leaked through I/O by remote attackers. Third, 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. 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 computers. As shown in §6, BUDDY only imposes a small performance overhead.

Although BUDDY leverages the n-version approach [18], *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—stopping memory disclosures*. To this end, we have proposed multiple new mechanisms such as semantic-preserving and divergence-triggering diversification. We differentiate BUDDY from many previous n-version systems [16], [17], [19], [20], [21], [22], [23], [24] 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 [16]) 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 [15]) should not be treated as a leak if it does not leave the OS through I/O write. ReMon [25] improves the efficiency by eliminating the synchronizations of non-sensitive syscalls; however, it does not provide divergence-triggering diversification for detecting memory disclosures that are based on relative addresses. BUDDY 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, BUDDY employs two diversification schemes based on a formal model to reliably detect memory disclosures caused by memory errors.

We have implemented BUDDY on the Linux 64-bit plat-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

form. BUDDY mainly consists of (1) diversifiers that diversify target programs in a semantic-preserving and divergence-triggering manner, (2) a kernel space coordinator that virtualizes and synchronizes syscalls, (3) a user space coordinator that virtualizes and synchronizes virtual syscalls and instructions, (4) a controller that prepares both coordinators, starts and monitors the buddy instances. We have evaluated the performance of BUDDY with the SPEC CPU2006 benchmarks and multiple server programs, including Apache web server, Nginx web server, OpenSSL, PHP, Lighttpd, and Orzhttpd. For effectiveness evaluation, we have evaluated BUDDY against a set of real memory disclosure attacks such as the HeartBleed attack. The evaluation results show that BUDDY can effectively prevent all tested memory disclosures. 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 benchmarks and around 4% on web server programs. Moreover, the diversification schemes introduce 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, our work makes the following contributions:

- We propose a detection methodology for memory disclosures. No matter what the cause is, the detection generally captures memory disclosures when they actually occur. The expensive and error-prone data-tracking process is avoided.
- We develop BUDDY, a system that realizes the detection methodology. Its core is an efficient replicated execution engine that employs the adaptive ring buffer-based coordination mechanism and a single-point synchronization mechanism.
- We develop a formal model for diversification and then design two diversification schemes to stop memory disclosures caused by memory errors.

1.1 Threat Model and Problem Scope

In this paper, we focus on preventing remote attacks that require memory disclosures to circumvent ASLR or steal sensitive data. After memory disclosures, the attacker may further launch subsequent attacks such as code-reuse attack. We assume the hardware and OS kernel as our trusted computing base (TCB), so attacks that target the hardware (e.g., cold boot attack [26]) 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 with privileged code execution abilities. Complementarily, researchers have proposed a significant number of approaches to detecting and preventing side-channel attacks, such as cache partitioning [27], cache and timing randomization [28], and system abnormal-pattern detection [29]. Instead, BUDDY focuses on preventing high-volume memory disclosures caused by memory errors, such as HeartBleed [5].

For the target program, we assume it may contain one or more vulnerabilities that can be exploited to disclose memory. While safe system programming languages such as Rust are available, most existing server and system programs are still written in unsafe languages, suffering from memory-disclosure vulnerabilities. We also do not limit the types of vulnerabilities (except side-channels); they can be buffer overreads, format strings, reading uninitialized memory, etc. We assume the source code of the target program is available so that we can recompile it to enforce the diversification. 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.

In the rest of paper, we compare BUDDY with related work (§2, describe the design of BUDDY in §4 and its implementation in §5, provide the security analysis of BUDDY and present evaluation results in §6. Finally, we discuss limitations of BUDDY in §7 and conclude in §8.

2 Related Work

Address Protection. Readactor [7], Binary Stirring [34], and ASLR-Guard [8] prevent code address leaks 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 [35] uses a single key to `XOR` all pointers, which is vulnerable to chosen-plaintext attacks [8]. BUDDY does not suffer from these issues.

To mitigate the impact of address leak, many proposed approaches [34], [36], [37], [38] improved ASLR in granularity. These fine-grained randomizations have been shown to be vulnerable to memory disclosure attacks [4]. Several re-randomization approaches [9], [39], [40] have been proposed to improve ASLR in frequency. OS-ASR [39] requires kernel modification. TASR [9] does not re-randomizes data sections. A general problem with such re-randomization approaches is the significant performance overhead. In comparison, BUDDY is a general memory disclosure detection system that stops leaks of both code and data addresses, and beyond.

N-Version and Multi-Execution. Many systems leverage n-version or multi-execution techniques to achieve security or privacy goals. Table 1 summarizes differences between BUDDY and representative n-version and multi-execution systems. Specifically, *n-variant systems* [16] an DCL [30] focus on preventing control flow hijacking attacks via replicated execution with partitioned address space. Both impose a significant performance overhead because they synchronize all syscalls. Varan [19] is an efficient and general n-version execution framework. It does not aim to stop attacks, thus does not strictly synchronize syscalls, allowing asymmetrical attacks [30]. TightLip [31] and Shadow execution [32] 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 flows dependent on inputs. Shadow execution [32] achieves shadow execution at virtual-machine level, which dramatically enlarges the TCB and increases the performance overhead. ReMon [25], MvArmor [22], and LDX [23] improve the efficiency of replicated execution. However, ReMon [25]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

System	Synchronization		Multi-execute	False	Overhead		Kernel	Addr.	Over-	Defeat
	strict/loose	target		report	(SPEC)	(latency)	change	leak	read	ROP
N-variant [16]	strict sync	all syscalls	user-space	high	N/A	17.8%	Yes	✗	✗	✓
Varan [19]	loose sync	all syscalls	user-space	low	14.2%	2.4%	No	✗	✗	✗
DCL [30]	strict sync	most syscalls	user-space	high	6.37%	N/A	No	✗	✗	✓
TightLip [31]	strict sync	all syscalls	user-space	high	N/A	5%	<1K	✓	✗	✗
Shadow Exe. [32]	strict sync	all inputs	user-s./VM	high	N/A	>100%	VM	✓	✗	✗
ReMon [25]	strict sync	sensitive syscalls	user-space	low	3.1%	2.4%	97	✓	✗	✓
MvArmor [22]	strict sync	syscalls	user-space	low	9%	55%	Yes	✓	heap	✓
LDX [23]	loose sync	outputs	user-space	high	4.7%	N/A	No	✓	✗	✓
Detile [33]	strict sync	interpreter	user-space	low	N/A	17%	No	✓	✗	✓
BUDDY	strict sync	I/O write	user-space	low	2.34%	4%	600	✓	✓	✓

TABLE 1: Comparison with representative n-version or multi-execution systems. `Defeat ROP` shows if the system can prevent code-reuse attacks. `Kernel change` shows if it changes OS kernel or virtual machine. For some cases, the number of line of added kernel code is available.

currently does not detect memory disclosures caused by out-of-bound reads based on relative addresses. MvArmor [22] cannot detect out-of-bound reads resulted from the stack, and its performance overhead is still significant. LDX [23] mutates inputs to detect divergences, which cannot detect out-of-bound overread. In contrast, BUDDY can efficiently and generally detect various memory disclosures by properly enforcing new diversification schemes. Detile [33] detects only address leaks in the scripting environment. Since it has to instrument call and return bytecodes in interpreter, which are executed frequently, it imposes a performance overhead of 17%. Secure multi-execution [41] is an approach for controlling illegal information flows from sensitive inputs instead of detecting memory disclosures. Its main idea is execute a program multiple times, once for each security level with inputs and outputs assigned to the same level, thus achieving non-interference. Ochoa et al. [42] discussed a similar idea of using multi-execution to prevent memory disclosures with the diversification of random canaries. However, they did not implement such a system. BUDDY is a efficient system equipped with diversification schmes with the divergence property (§4.1.1).

BUDDY leverages the concept of n-version programming for a new goal—detecting memory disclosures, and employs a set of new techniques to adapt the new scenario. 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; (2) false positives, some internal divergences are normal and should not be reported as memory disclosures. Unlike traditional n-version systems, BUDDY reports divergence only at I/O write—when the protected data actually leaves the process. This design choice minimizes the performance overhead and false positives. To faithfully detect memory disclosures, we further develop a formal model and two schemes for diversification so that BUDDY can generally detect memory disclosures in a principled manner.

Memory Corruption Preventions. Memory errors can be exploited to leak data. Many memory safety approaches [11], [12], [17], [43], [44] provide a strong protection against a broad range of memory errors (e.g., out-of-bound read/write and use-after free). Memory safety approaches usually impose a significant performance overhead. For example, even the recent hardware-accelerated memory safety approach WatchdogLite [43] 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 information leaks caused by uninitialized reads [15].

3 THE BUDDY APPROACH

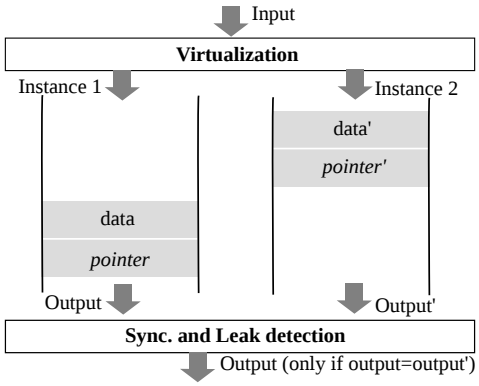

Fig. 1: Overview of the BUDDY approach. BUDDY first diversifies the target program in a fine-grained manner, and then virtualizes all virtualizing points and strictly synchronizes at I/O write to detect any divergence in outgoing data. For example, If $pointer$ and $pointer'$ are disclosed, BUDDY is able to detect it at socket write because they are not equal.

A memory disclosure occurs when unintendedly read data is sent to the external world. We observe that memory disclosures are consequences of a number of hard-to-detect program errors. Therefore, a general and intuitive way to detect memory disclosures is to check whether the outgoing data contains any target data. However, this is impractical because we do not know how the target data looks like—data may be computed and transformed into other forms during the execution. To solve this problem, we propose the replicated execution–based detection. We transform the detection problem into an equivalence-check problem by seamlessly maintaining two identical instances of a target process and diversifying only the target data. We can always

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

detect any disclosure of such data because the disclosure will always result in the two instances outputting different values. This way, we are able to generally capture any memory disclosures without the need of tracking the propagation of the target data.

Figure 1 illustrates how BUDDY works: BUDDY first diversifies the target program in a way that semantics are preserved and memory disclosures will be detected when they are triggered. The diversification schemes ensure that reading the target data in these buddy instances through memory errors will get different values. When the target program is about to run, BUDDY then launches two identical instances (processes). 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 it *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) diversification schemes that diversify the target data but will not break program semantics, (2) virtualization of buddy instances, which properly intercepts external inputs and non-determinisms to ensure that they always return the same value, and (3) disclosure detection that captures divergences at I/O write.

4 DESIGN

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

4.1 Semantic-Preserving and Divergence-Triggering Diversification

In order to diversify buddy instances in a principled manner, we first develop a formal model, based on which, we then design two diversification schemes.

4.1.1 *Formalization of* BUDDY

BUDDY aims to detect attacks that exploit memory errors to disclose memory data. 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 produce 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:

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

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

The above equations shows the following property, namely `equivalence property`, under normal execution when we assume that 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 differ. 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.

$$\forall 0 \leq i \leq N, \forall I \in Inputs: \\ T(S_i, I)) \ or \ T'(S'_i, I)) \in Memory\ disclosures \\ \implies Map(S_{i+1}) \neq Map(S'_{i+1}). \quad (3)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

4.1.2 Diversification Schemes

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 in outputs. Further, the diversification scheme should also satisfy the `divergence property`, i.e., any memory disclosures can be detected. Diversifying the target data itself is ideal to memory disclosure detection. However, since program semantics often depends on data, diversifying data will likely change program semantics, dissatisfying the `equivalence property`. In this work, we design diversification schemes that do not diversify the target data but, in most cases, satisfy both the `equivalence property` and the `divergence property`. Specifically, 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 [17], which allocates objects in random addresses. Note that, BUDDY is a general system; more diversification schemes can be applied to achieve other security properties.

Partitioned Address Randomization. Inspired by address space partitioning [16], we develop the `partitioned address randomization` to detect absolute address-based over-reads. Note that this scheme is similar to address space partitioning [16]; the only difference is its self-randomization that mitigates asymmetrical attacks [30]. Specifically, 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 further retrofit the loader (`ld.so`) to ensure that memory layout randomization is also enforced separately in each sub-space.

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. Figure 2 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.

For stack, the padding is inserted between stack frames (right after return address). The x86_64 calling convention passes the first six parameters through registers, but the remaining ones are still passed through stack. As such, we also need to update all the parameter-accessing instructions by patching their 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) with compiler supports; however, it will incur more performance overhead. We currently choose object-level and stack-frame-level padding for a better performance.

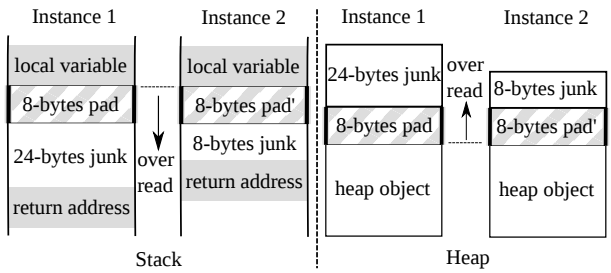

Fig. 2: The random padding scheme for stack and heap objects. A random value is inserted between any two stack frames or heap objects. The size difference between placeholders ensures different offsets.

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, the equivalence property holds 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 [5]), 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

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 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. This way, we ensure that BUDDY behaves as a single entity to attackers, and whenever an divergence is detected, we conclude with confidence that memory disclosures occur.

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). In BUDDY, we ensure virtualizing points always 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 has been handled by all n-version systems [16], [17], [19], [45]. Similar to these previous systems, we also conservatively include most syscalls, all virtual syscalls, and non-deterministic instructions as the virtualizing point set.

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. For syscalls, we choose to temporarily patch the syscall table using a kernel module. This way, extra context switches can be avoided to improve performance. For virtual syscalls, we choose to patch the `GOTPLT` table that contains the entries to these virtual syscalls. For non-deterministic instructions such as `RDTSC` and `RDRAND`, we replace them with an one-byte interrupt instruction (e.g., `INT3`) which allows us to perform virtualization by handling the interrupt.

The Adaptive Ring Buffer Coordination. Traditional n-version systems perform coordination in a fully synchronized manner (i.e., lockstep), which introduces 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 borrow the ring buffer idea from Varan [19] and RBNB [46]. 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. Because the leader does not need to wait for the follower, the performance overhead is much lower. In order to prevent memory disclosures, we further enforce synchronization to I/O write operations and check for divergences in the outgoing data.

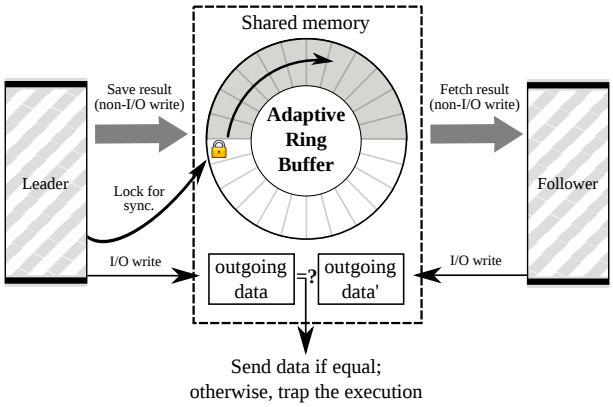

Fig. 3: The adaptive ring buffer-based virtualization. Normal virtualizing points are virtualized in a ring buffer manner. However, upon I/O write, BUDDY locks the ring buffer and strictly synchronizes to detect memory disclosures.

To implement our coordination scheme, we designed the *adaptive ring buffer*-based virtualization mechanism (Figure 3). 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. Attackers 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 interleaving is another important source of non-determinism for multi-threaded programs. Divergences in thread interleaving in the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

two instances will result in "benign" behavior divergences (e.g., divergent I/O write system calls) in the replicated execution, causing false postives in detecting memory disclosures. To completely resolve this issue, synchronizing all shared resource/memory accesses is necessary to ensure that the leader and follower have consistent views on shared resource/memory. 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) [47], [48], an orthogonal domain to BUDDY. The state-of-the-art mechanism for synchronizing multi-threaded variants was proposed by Volckaert et al. [49]. Their mechanism records, in the leader instance, system calls operating on shared resources and synchronization-variables (e.g., locks and condition variables) accesses, and enforces them to be executed in the same order in the follower instance. This way, this mechanism can eliminate divergences caused by multithreading.

In BUDDY, similar to Varan [19] and the system-call synchronization part of [49], we implement 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 to synchronize system calls. 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.

BUDDY does not further synchronize accesses to synchronization-variables in threads as in [49]. In fact, as also mentioned in [49], 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 (§6).

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 compare 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.

5 IMPLEMENTATION

We implemented BUDDY on 64-bit Linux, which consists of diversifiers, a kernel space coordinator, a user space coordinator, and an instance controller.

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 `mmap`.

Random padding diversifier. We insert random paddings among stack frames and heap objects. For stack random padding, if source code of the target program is available, we can leverage compilers to insert padding spaces and update offsets in parameter-accessing instructions. In current implementation, we modify the GNU assembler. Such implementation has the potential to support COTS binaries as long as the disassembly is complete and reassembleable [50]. Specifically, the program is compiled with flags `-pie -fPIC -O2 -fno-omit-frame-pointer`. 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 through the pattern "`offset(%rbp)`." If `-fno-omit-frame-pointer` is not specified during compilation, we can instead use the pattern "`offset(%rsp)`." However, in this case we need to further analyze the value of `offset` to differentiate parameter-accessing instructions from local variable–accessing instructions based on the boundary of the stack frame. We then 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. Note that such implementation does not detect internal overreads across objects in the same allocation.

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 detection. Syscall

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

interception is done through standard syscall table patching, i.e., replacing the original syscall handlers with our callbacks.

I/O write system calls. Any operation that may "send" memory to the external should be intercepted for divergence checks. These operations can be roughly classified into three categories. (1) I/O write system calls: They are high-volume and the most common operations for disclosing memory to the external. (2) Mapped-memory write operations: A common example is memory-mapped I/O write in which no system call is required for writing. (3) Implicit disclosure channels: Disclosed memory data is used as observable side information such as file pathnames. Current implementation of BUDDY covers only the first category. Nine system calls (`send`, `sendto`, `sendmsg`, `sendmmsg`, `write`, `writev`, `pwrite64`, `pwritev`, and `pwritev2`) are currently intercepted for divergence checks. To include the second category, we will have to analyze the code of the target program to identify mapped memory (e.g., through `mmap` with parameters `PROT_WRITE` and a file descriptor) that is accessible to the external and all writes to these memory regions. These writes should also be intercepted for divergence checks. The third category implicitly disclose memory through side information that is observable to external, e.g., using the disclosed memory data as a pathname to create a file. Such memory disclosures are hard to exploit without hijacking the data flow. Common implicit channels such as pathnames should be included for divergence checks; however, it is hard to include all cases, which is a limitation of BUDDY.

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 first informs 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.

5.2.1 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. Interception is implemented as depicted in §4.2. Two special cases are (1) getpid() becomes a virtual syscall after its first call, so we also need to intercept through patching the GOTPLT table; (2) to intercept `RDTSC`, instead of rewriting the binary, we trap for RDTSC by making timestamp counter non-readable (i.e., prctl(PR_SET_TSC, PR_TSC_SIGSEGV, 0, 0, 0)).

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.

5.2.2 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.

6 EVALUATION

In this section, we extensively evaluate the robustness, security, and performance of BUDDY.

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. All the tested programs are compiled with optimization level `-O2` with flags `-pie -fPIC -fno-omit-frame-pointer`.

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 does 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 [51], which contain HTML and embedded scripts. We then configured the servers to host them. After that, we ran ApacheBench [52] 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 [53], it is insecure and should be avoided at best. After compiling OpenSSL with the `-DPURIFY` flag (not using uninitialized buffers as a source of randomness), BUDDY functions correctly without any false positive.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

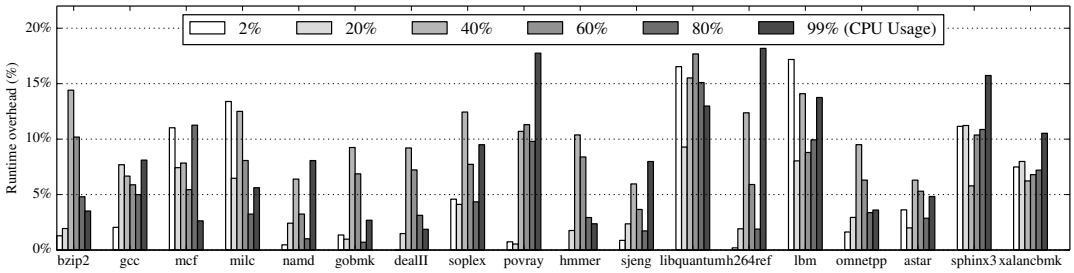

Fig. 4: Performance of SPEC benchmarks when CPU is in various load levels. CPU usage is controlled by `stress-ng`. When CPU is highly-loaded (99% usage), BUDDY imposes an average performance overhead of 8.3%. Numbers are measured over all cores.

6.2 Security

BUDDY is secure because: (1) in principle, if any "leaked" data is useful for the attacker to infer the randomized address, it must be differential; otherwise, it does not contain meaningful entropy. (2) unlike encryption-based approaches [35], [54] whose security depends on the confidentiality of the key used for encryption, BUDDY is a keyless system. The security of BUDDY does not rely on keeping any secret keys. We validate the security of BUDDY by testing it against 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 [55] showed that a format string vulnerability in Orzhttpd [1] 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 [56] showed that ASLR can be bypassed within a few minutes. Its core idea is to leverage a simple stack buffer overflow vulnerability to 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[2] 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.

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[3]. 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.

6.3 Performance of the BUDDY Framework

In this section, we first measure the performance of BUDDY with the partitioned address randomization scheme enabled. We then evaluate the performance of the random padding scheme. Note that all performance numbers are measured over *a pair of buddy instances*, not an individual instance. The baseline performance is measured based on the unpatched kernel without loading our kernel module.

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.

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 5 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 [7] that protects code pointer only and requires hardware assistance. BUDDY is significantly more efficient than previous n-version systems. We believe the performance gain is from the fact that BUDDY synchronizes 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`[4] 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 4. The low overhead on SPEC benchmarks indicates that BUDDY is also a practical solution in protecting CPU intensive programs.

Performance impact on other processes. The PID-based process filtering scheme used in BUDDY requires that syscalls from all processes running will be first hooked by the BUDDY

1. http://code.google.com/p/orzhttpd/source/detail?r=141
2. http://www.scs.stanford.edu/brop/
3. https://www.exploit-db.com/exploits/32998/
4. http://kernel.ubuntu.com/~cking/stress-ng/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

Level	c = 1			c = 16			c = 64			c = 256			
Worker	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Ave.
p=1	16.2	17.2	(6.4%)	15.2	16.8	(10.6%)	14.5	16.1	(10.6%)	13.6	14.8	(9.0%)	9.2%
p=2	15.5	17.2	(10.5%)	10.1	10.5	(3.8%)	9.8	10.7	(9.2%)	9.7	10.4	(6.4%)	7.5%
p=3	16.2	16.6	(2.2%)	9.7	9.7	(0.1%)	9.5	9.7	(2.3%)	9.4	9.7	(2.4%)	1.7%
p=4	15.9	17.1	(7.9%)	9.8	10.0	(2.4%)	9.5	9.7	(1.9%)	10.5	11.0	(5.0%)	4.3%
p=5	17.1	17.4	(1.6%)	9.4	9.5	(0.3%)	9.7	9.8	(1.2%)	10.6	11.2	(4.7%)	2.0%
p=6	16.1	17.4	(8.1%)	9.3	9.7	(4.1%)	9.7	9.8	(1.8%)	10.7	11.8	(10.8%)	6.2%
p=7	15.2	16.7	(9.7%)	9.2	9.5	(3.5%)	9.8	10.1	(2.2%)	11.2	12.3	(10.5%)	6.5%
p=8	16.6	16.6	(0.0%)	9.4	10.1	(7.7%)	10.6	10.9	(2.5%)	12.4	13.1	(5.9%)	4.0%
Geomean			4.4%			2.1%			2.9%			6.2%	3.6%

TABLE 2: Overhead of processing time (ms) per request incurred to Apache httpd under BUDDY with various numbers of worker processes and workloads. s=1MB.

Level	c = 1			c = 16			c = 64			c = 256			
Server	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Ave.
Apache	63.0	58.3	(7.4%)	102.4	100.0	(2.4%)	105.1	103.2	(1.8%)	95.6	91.0	(4.7%)	4.1%
Nginx	64.0	60.7	(6.5%)	55.2	52.7	(4.6%)	52.6	51.1	(2.8%)	31.6	30.1	(4.7%)	4.7%
Lighttpd	63.3	60.3	(4.8%)	58.1	54.9	(5.5%)	59.2	54.8	(7.5%)	40.3	37.6	(6.8%)	6.2%
PHP	64.9	64.7	(0.3%)	58.7	54.2	(7.7%)	44.5	42.9	(3.6%)	28.2	26.9	(4.5%)	4.0%
Geomean			2.9%			4.6%			3.4%			5.1%	3.9%

TABLE 3: Slowdown of number of requests per second incurred to popular web servers with and without BUDDY under various workloads. p=4, s=1MB.

Level	s = 1KB			s = 256KB			s = 1MB			s = 16MB			
Server	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Orig.	BUDDY	(%)	Ave.
Apache	2.3	2.2	(3.4%)	78.7	71.8	(8.7%)	102.4	100.0	(2.4%)	111.1	105.1	(5.3%)	5.0%
Nginx	2.8	2.6	(5.3%)	17.2	17.0	(1.4%)	55.2	52.7	(4.6%)	96.5	90.1	(6.6%)	4.5%
Lighttpd	3.3	3.0	(6.6%)	17.4	16.9	(3.4%)	58.1	54.9	(5.5%)	105.6	99.1	(6.2%)	5.4%
PHP	2.4	2.2	(6.9%)	17.3	16.4	(5.6%)	58.7	54.2	(7.7%)	93.1	91.3	(1.9%)	5.5%
Geomean			5.4%			3.9%			4.7%			4.5%	4.6%

TABLE 4: Overhead of rate of data transfer (MB/s) incurred to popular web servers with and without BUDDY when serving various sizes of resource. p=4, c=16.

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**%	

TABLE 5: Performance evaluation of BUDDY on the SPEC2006 benchmarks

kernel module for PID checking in order to filter out non-BUDDY-protected processes. This instrumentation adds on overhead on almost all syscall execution paths which might negatively impact the performance of all other processes.

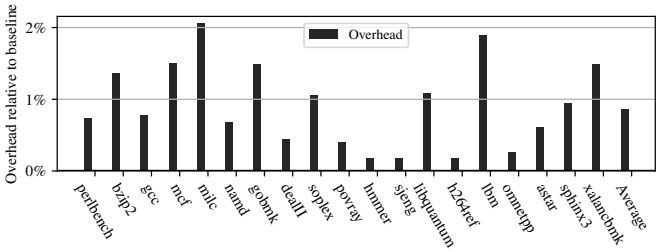

Fig. 5: Overhead of BUDDY on other processes in the system (i.e., processes not protected under BUDDY). On average, The BUDDY kernel module imposes an overhead of less than 1%.

To evaluate this overhead, we measure the performance of the SPEC benchmarks with and without loading the BUDDY kernel module and report the overhead in Figure 5. The results show that on average, BUDDY adds less than 1% overhead to SPEC programs with the worst case a bit higher than 2% (program milc). We believe this is a tolerable overhead for the overall system.

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 pro-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

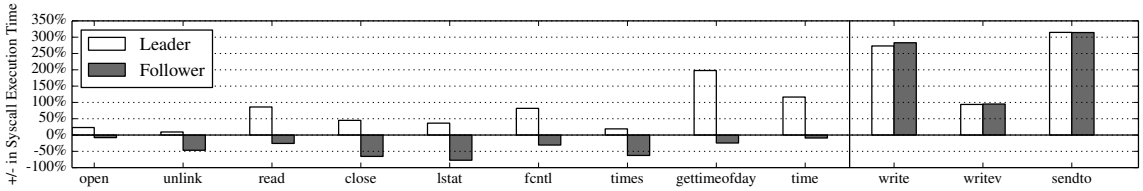

Fig. 6: Micro-benchmarks on syscall execution. A negative overhead could only happen to the follower instance.

cesses (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 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 2 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 the 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 results of this experiment show is shown in Table 3, and the average overhead is less than 8%. To measure the impact of different *file size*, we set the concurrency to 16. Table 4 shows that the average overhead is also less than 8%. Based on these results, we believe that BUDDY is a practical tool for protecting server programs.

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 6). 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.

6.3.1 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 §6.3, and the results are also shown in Table 5. 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.

7 DISCUSSION

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.

Possible attacks. The possible attacks [30] against BUDDY are (1) control attacks against one instance, (2) memory overwriting one instance to unify the divergences, and (3) compromising the monitor of BUDDY to disable divergence checks. Attack (1) needs to first leak randomized address of the instance to launch meaningful control attacks, which is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2878234, IEEE Transactions on Dependable and Secure Computing

however prevented by BUDDY. Attack (2) needs to overwrite the memory in the first instance to match the diversified memory in the second instance, which requires information leak of the memory in the second instance. Such an information leak is also hard because of BUDDY's protection. In BUDDY, we assume the hardware and the OS kernel are trusted. To launch attack (3), attackers need to exploit vulnerabilities in the monitor of BUDDY, which is possible in practice.

Supporting script environments. The detection mechanism of BUDDY is general—two 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. Therefore, it is possible to extend BUDDY to support script environments such as the JavaScript engine in browsers. 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. So to prevent attacks from scripts, we either need to disable JIT or check every memory read; but either way will introduce a significant performance overhead.

8 CONCLUSION

Memory disclosures break not only the confidentiality of secret data but also the effectiveness of security mechanisms such as ASLR. Unfortunately, we still lack a general and practical detection or defense mechanism. In this paper, we have proposed a general methodology for detecting memory disclosures and developed BUDDY to realize the methodology. BUDDY enforces semantic-preserving and divergence-triggering diversification to the target program, and maintains two running instances (one original and the other one diversified). Any memory disclosure will result in the two instances outputting different data values, triggering the divergence detection. No matter what the cause is and how data is propagated (even through dynamically generated code), BUDDY can generally detect a memory disclosure without the need of performing the error-prone and expensive data-flow tracking. Extensive evaluation results show that BUDDY can effectively stop real memory disclosures (e.g., HeartBleed) with a small performance overhead on multi-core processors.

REFERENCES

[1] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "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.

[2] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *22nd ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[3] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[4] 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 *22nd Annual Network & Distributed System Security Symposium*, 2015.

[5] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, 2014.

[6] C. Details, "Vulnerabilities By Type," 2015, http://www.cvedetails.com/vulnerabilities-by-types.php.

[7] 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 *36th IEEE Symposium on Security and Privacy*, 2015.

[8] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "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.

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

[10] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[11] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[12] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler Enforced Temporal Safety for C," in *International Symposium on Memory Management (ISMM)*, 2010.

[13] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[14] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[15] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, Oct. 2016.

[16] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *In Proceedings of the 15th USENIX Security Symposium*, 2006.

[17] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006.

[18] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Digest FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, June 1978.

[19] P. Hosek and C. Cadar, "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*, ser. ASPLOS '15, 2015.

[20] M. Maurer and D. Brumley, "Tachyon: Tandem execution for efficient live patch testing," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[21] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance for intrusion detection," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05, 2006.

[22] K. Koning, H. Bos, and C. Giuffrida, "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, Jun. 2016.

[23] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Mar. 2016.

[24] B. Salamat, T. Jackson, A. Gal, and M. Franz, "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*, ser. EuroSys '09, 2009.
[25] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *Proceedings of the 2016 ATC Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.
[26] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," in *Proceedings of the 18th Usenix Security Symposium (Security)*, Montreal, Canada, Aug. 2009.
[27] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41, 2008.
[28] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *22nd Annual Network & Distributed System Security Symposium*, 2015.
[29] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *19th International Symposium on Research in Attacks, Intrusions, and Defenses*, ser. RAID'16, 2016.
[30] S. Volckaert, B. Coppens, and B. De Sutter, "Cloning your gadgets: Complete rop attack immunity with multi-variant execution," *IEEE Transactions on Dependable and Secure Computing*, 2015.
[31] A. R. Yumerefendi, B. Mickle, and L. P. Cox, "Tightlip: Keeping applications from spilling the beans," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI '07, 2007.
[32] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla, "Preventing information leaks through shadow executions," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, ser. ACSAC '08, 2008.
[33] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, San Sebastián, Spain, Jul. 2016.
[34] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.
[35] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard TM: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.
[36] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd USENIX Security Symposium*, Aug. 2014.
[37] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Proceedings of the 22Nd Annual Computer Security Applications Conference*, 2006.
[38] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.
[39] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "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.
[40] K. L. S. N. M. Backes and W. Lee, "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.
[41] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.
[42] M. Ochoa, S. Banescu, C. Disenfeld, G. Barthe, and V. Ganesh, "Reasoning about probabilistic defense mechanisms against remote attacks," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017.
[43] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, 2014.
[44] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
[45] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replica for defeating memory error exploits," *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 2007.
[46] S. Tilak, P. Hubbard, M. Miller, and T. Fountain, "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*, ser. E-SCIENCE '07, 2007.
[47] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "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*, ser. ASPLOS XV, 2010.
[48] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, Mar. 2009.
[49] S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, 2017.
[50] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.
[51] Alexa, "Alexa Top 500 Global Sites," 2016, http://www.alexa.com/topsites.
[52] Apache, "ab - Apache HTTP server benchmarking tool," 2016, https://httpd.apache.org/docs/2.2/programs/ab.html.
[53] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991.
[54] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.
[55] *Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks*, San Jose, CA, May 2016.
[56] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

Kangjie Lu is an assistant professor in the Computer Science & Engineering Department of the University of Minnesota-Twin Cities. He receivedd the Ph.D. degree in Computer Science from the Georgia Institute of Technology. His current research aims to secure computer systems by hardening code and design, finding vulnerabilities, and detecting privacy leaks.

Meng Xu is a Ph.D. candidate in Computer Science at the Georgia Institute of Technology, advised by Professor Taesoo Kim. His research interests include systems security, N-version programming, and bug finding.

Chengyu Song is an assistant professor in the department of Computer Science and Engineering at University of California, Riverside. He received the Ph.D. degree in Computer Science from the Georgia Institute of Technology. His research interests include system security, program analysis and verification, and operating systems.

Taesoo Kim a Catherine M. and James E. Allchin Early Career Assistant Professor in Computer Science at the Georgia Institute of Technology. He also serves as the director of the Georgia Tech Systems Software and Security Center (GTS3). He is genuinely interested in building systems that prioritize security principles first and foremost.

Wenke Lee is a professor in Computer Science at the Georgia Institute of Technology. He is the Co-Director of the Institute for Information Security & Privacy at Georgia Tech. He received the Ph.D. degree in computer science from Columbia University. His research interests include systems and network security, applied cryptography, and machine learning.