ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations

Koustubha Bhat
Vrije Universiteit Amsterdam
The Netherlands
k.bhat@vu.nl

Erik van der Kouwe
Leiden University
The Netherlands
e.van.der.kouwe@liacs.leidenuniv.nl

Herbert Bos
Vrije Universiteit Amsterdam
The Netherlands
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
The Netherlands
giuffrida@cs.vu.nl

Abstract

Many modern defenses against code reuse rely on hiding sensitive data such as shadow stacks in a huge memory address space. While much more efficient than traditional integrity-based defenses, these solutions are vulnerable to probing attacks which quickly locate the hidden data and compromise security. This has led researchers to question the value of information hiding in real-world software security. Instead, we argue that such a limitation is *not* fundamental and that information hiding and integrity-based defenses are two extremes of a continuous spectrum of solutions. We propose a solution, ProbeGuard, that automatically balances performance and security by deploying an existing information hiding based baseline defense and then incrementally moving to more powerful integrity-based defenses by hotpatching when probing attacks occur. ProbeGuard is efficient, provides strong security, and gracefully trades off performance upon encountering more probing primitives.

CCS Concepts • **Security and privacy → Systems security**; *Software security engineering*.

Keywords reactive defenses, program transformations, hotpatching, processor trace, security hardening, performance-security tradeoff, graceful performance degradation, information hiding, code reuse, software bugs

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304073

ACM Reference Format:
Koustubha Bhat, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2019. ProbeGuard: Mitigating Probing Attacks Through Reactive Program Transformations. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304073

1 Introduction

Today's memory corruption attacks routinely bypass defenses such as Data Execution Prevention (DEP) by means of reusing code that is already in the program [55]. To do so, attackers need knowledge of the locations of recognizable code snippets in the application's address space for diverting the program's control flow toward them.

Rigorously enforcing software integrity measures throughout an application such as, bounds-checking on accesses to buffers and data structures in memory, control flow integrity checks that ensure application behavior remains within the program's intended control flow, thwarts such attacks but, at a steep cost in performance [9, 21, 49–51, 58, 61]. To illustrate, we can expect applications to incur approximately an average slowdown of up to 9% to enforce forward-edge Control Flow Integrity (CFI) [61] that protects calls to functions, then 3.5 - 10% for shadow stacks to protect backward-edges [20] (protecting returns from functions), further 4% to prevent information leakage and 19.6% to thwart data corruption attacks by restricting memory reads and writes in the application through Software Fault Isolation (SFI) [42, 65]. Clearly, all these defenses combined to counter many classes of attacks incur a non-trivial cumulative overhead. Yet, a sufficient defense may need even more integrity measures.

An alternative to such solutions that enforce software integrity, is to make it difficult to *locate* code and data in the first place. Examples of this approach range from simple address space layout randomization (ASLR), to advanced defenses that hide sensitive information at random locations in a large address space [18, 44, 47]. For instance, Code Pointer

Integrity [44] moves all sensitive data such as code pointers to a "safe" region at a hidden location in memory. As a defense, such *information hiding* is more efficient than *integrity-based defenses* [8]. In particular, randomization is almost 'free', as even a sophisticated defense against code reuse attacks such as Code Pointer Integrity (CPI) adds a modest 2.9% performance overhead.

Unfortunately, recent research demonstrates that attackers bypass even the most advanced information-hiding defenses [14, 15, 25, 31, 33, 39, 54]. They show that, by repeatedly 'probing' the address space (either directly or by means of side channels), it is possible to break the underlying randomization and reveal the sensitive data. With this, even a robust information-hiding based defense stands defeated.

Thus, to protect against modern attacks, developers face an awkward dilemma: should they employ software integrity measures that are strong but very expensive (perhaps prohibitively so), or defenses based on information hiding that are fast, but offer weak protection? In this paper, we show that we can combine the best of both—by transitioning from fast information hiding to strong software integrity if (and only when) attackers start probing to break the randomization.

Derandomization primitives To break randomization, attackers make use of a number of derandomization primitives. Examples include crashing reads and jumps [14], their crash-less counterparts [25, 31], and employing allocation oracles [54] among others. Since one-shot leaks are rare in modern defenses—as the defenses move all sensitive information (e.g., code pointers) out of reach of the attacker, state-of-the-art derandomization primitives invariably must *probe* by repeatedly executing an operation (e.g., a memory read) to exhaust the entropy. As there is no shortage of primitives, it is tempting to think that information hiding is doomed and integrity solutions are the future.

Instead, we argue that being vulnerable to probing in itself is not a fundamental limitation. We see information hiding and integrity-check defenses as two extremes of a continuous spectrum of defenses against code reuse attacks, where they trade off between efficiency and security. Information hiding can still hold its ground if a system could detect the probing process and stop it before it breaks the defense.

Selective hardening The key idea we present is that, in a software protected by a fast baseline defense (information hiding), we keep monitoring the running program for any occurrence of probing attempts. When we encounter any such attempt, we automatically locate its origin, and patch *only* the offending piece of code at runtime with stronger and more expensive integrity-based defenses. In other words, we apply strong defenses selectively, as needed—resulting in strong protection within low overheads.

The first stage of ProbeGuard is a form of anomaly detection. We detect probing attempts that characterize derandomization primitives. However, unlike traditional anomaly detection, false positives are less of a problem. They merely lead to more hardening of part of the program to make it more secure, albeit somewhat slower. For most varieties of probing attacks, the anomaly detection itself is simple and non-intrusive (for example, a monitor detecting repeated exceptions or other anomalies).

The second stage, namely probe analysis, uncovers the particular code site the attacker abused for probing, or simply put, the *probing primitive*. Doing so is complicated in the general case. However, by leveraging fast control-flow tracing features available in modern processors (such as Intel Processor Trace (Intel PT) [38]), ProbeGuard conservatively pinpoints the offending code fragment in a secure way.

Finally, in the third stage, ProbeGuard hotpatch the program by selectively replacing the offending code fragment with a hardened variant, a strategy inspired by prior work on *hotpatching* (also known as *live* or *dynamic* software updating) [10, 26–29, 36, 48, 53]. Although now this piece of code runs slower, the instrumentation (and thus the slowdown) is limited to the fragment that was vulnerable. In principle, ProbeGuard is agnostic to the hotpatching technique itself. A simple and elegant way is to create a binary that already contains multiple versions of all code fragments, where each version offers different levels of protection. Initially, the binary only runs efficient, instrumentation-free fragments. However, as and when the probe analysis exposes a code fragment used as a probing primitive, ProbeGuard switches the corresponding code fragment to an appropriate hardened version.

Contributions We make the following contributions:

1. We present a new point in the design space of code reuse defenses that automatically balances performance and security. The design initially protects the system using (fast but weak) information hiding, and selectively transitions to (stronger but slower) integrity defenses, where and when needed.
2. We show that low-overhead control-flow tracing capabilities in modern processors (such as Intel PT) allow us to efficiently pinpoint code fragments affected by the probing attempts.
3. We demonstrate the notion of *reactive defense*, by making use of anomaly detection to trigger selective security hardening, in our open-source prototype system, ProbeGuard. Our experimental evaluation shows that ProbeGuard is secure, efficient and effective at countering probing attempts for a broad range of derandomization primitives.

Attacker intent	Remote code reuse attack.
Attack target	Server applications with automatic crash-recovery.
Attacker powers	Derandomization primitives, unlimited probe attempts.
Trusted Computing Base	Baseline info-hiding defense, OS, hardware including Intel PT traces.

Table 1. Threat model

2 Threat Model

We define a threat model that is in line with related research in the recent past [25, 31, 42, 54, 56] (Table 1). We consider a determined remote attacker who aims to mount a code reuse attack over the network on a *server application* hardened by any *ideal* state-of-the-art information hiding-based defenses. For example, one can secure a server application against code reuse by deploying a modern defense such as Code Pointer Integrity (CPI) (including SafeStack) [44] or leakage-resistant variants such as Readactor [19]. ProbeGuard's goal is to address the fundamental weakness of practical (information hiding-based) code-reuse defenses, making them resistant to attacks that bypass the defense by derandomizing hidden memory regions (such as, *safe region* and *trampoline code area* in CPI and Readactor respectively). While ProbeGuard aims to guarantee that the fundamental assumption of information hiding continues to hold true, we do note that characteristics and other limitations inherent to the deployed code-reuse defense remain as-is.

We trust the underlying operating system (e.g., Linux) which we assume to have all standard defenses enabled. In addition, we assume a modern processor system that provides efficient control flow tracing, such as Intel Processor Trace which is available on Intel CPUs since Broadwell. The trace is accessible via the operating system kernel, beyond the reach of a remote application-level attacker.

We assume a determined attacker who has access to derandomization primitives [31, 39, 54] to probe the victim's address space, find sensitive defense-specific information and bypass the defense. While the probability of finding the sensitive data in a 64-bit address space by accident using a single probe is negligible, we do assume that the attacker has unlimited probing attempts as the application recovers automatically upon any crash. This is realistic because, even though probing attempts may each lead to a crash, real-world server applications typically have worker processes with built-in crash recovery functionalities to deal with unexpected run-time errors [54].

3 Background & Related Work

We first outline existing defenses, classifying them into *software integrity* checks based and *information hiding* based defenses. In particular, we reason about why, despite the weakness, the latter remains a preferred choice for practical deployment. We then describe recent attacks against information hiding-based defenses.

3.1 Software Integrity

Whether the target is information leakage or code reuse exploitation, memory corruption attacks typically violate software integrity. To prevent this, software integrity defenses apply integrity checks throughout the application.

Many information leakage defenses add pervasive spatial, temporal, or type checks on memory accesses [23, 35, 43, 51, 52, 58, 62]. Modulo optimizations, such solutions verify all the program's loads and stores, as well as its memory allocation operations. They vary in terms of efficiency (although they generally incur high overhead) and how they manage (trusted) metadata. To learn the right bounds information, they may also rely on sophisticated static program analyses such as alias analysis and pointer tracing and tend to be robust in the security guarantees they offer—except for the well-known (and fundamental) limitations of such analysis techniques. To counter code reuse attacks that modify the control flow of an application, solutions like Control Flow Integrity [8] (CFI) check each indirect control-flow transfer to see if it adheres to the application's static control-flow graph. Unfortunately, fine-grained CFI [21, 49, 50, 61] incurs significant performance costs and later variants [67, 68] therefore tried to balance security and performance guarantees. However, previous research has shown that doing so often significantly weakens security [30]. The overhead of fine-grained CFI can be as high as 21% [9] or as little as 9%, if we limit protection to the forward edge [61]. Finally, SFI (Software Fault Isolation) [42, 65], a sandboxing technique, that prevents arbitrary memory access or corruption, incurs about 17-19% overhead for both reads and writes.

3.2 Defenses based on information hiding

Defenses based on information hiding incur much less overhead as they eliminate expensive runtime checks and the integrity of hidden sensitive information rests solely on the attackers' inability to locate it. ASLR in particular serves as a first line of defense against code reuse attacks in many current systems. However, relying on ASLR alone is no longer sufficient when a variety of information disclosure vulnerabilities allow attackers to leak pointers to eventually break the randomization. Instead of merely hiding locations of entire applications, modern defenses therefore reduce the size of what remains hidden, by segregating applications into sensitive and non-sensitive regions and use probabilistic techniques based on ASLR to hide the sensitive regions.

Examples include CPI [44], Oxymoron [12], Isomeron [22], ASLR-Guard [47], and many others [13, 16, 17, 19, 60, 66]. CPI [44] hides a safe region and a safe stack where it stores all code pointers. ASLR-Guard [47] hides pre-allocated keys

Defense	*Arbitrary read*	*Arbitrary write*	*Arbitrary jump*	*Allocation oracle*
CCFIR	√			√
O-CFI	√			√
Shadow stack		√		√
StackArmor	√	√		√
Oxymoron	√	√	√	√
Isomeron	√	√	√	√
CPI	√	√		√
ASLR-Guard	√	√	√	√
LR2			√	√
Readactor			√	√

Table 2. Potential primitives (marked by √) to attack various *information hiding*-based defenses

that it uses for `xor`-based encryption of code pointers. Isomeron [22] and Oxymoron [12] hide runtime lookup tables to implement code randomization. All of them make code reuse infeasible by hiding sensitive data and eliminating the need for pervasive integrity checks. Among them, the leakage-resilient variants [13, 16, 17, 19, 47, 66] provide protection against JIT ROP [22, 59] attacks, by preventing attackers from reading executable code regions in memory.

All these techniques have very low runtime overheads. For instance, CPI-SafeStack reports less than 2% and ASLR-Guard reports less than 1%. Even if the security guarantees are less strong than integrity-based solutions, performance-wise, information hiding by means of randomization comes almost for free.

3.3 Attacks on information hiding

Unfortunately, information hiding is vulnerable to information disclosure. For instance, Evans et al. [24] attack the safe region of CPI by exploiting data pointer overwrite vulnerabilities, leaking the safe region's location through fault and timing-based side channels.

On a coarser level, Blind ROP (BROP) [14] exploits stack vulnerabilities to poke blindly into the address space and make the program jump to unintended locations. By observing the resulting crashes, hangs and other behaviors, attackers eventually find interesting gadgets—albeit after many crashes. CROP [25], on the other hand, abuses reliability features such as exception handling to prevent a crash upon probing inaccessible memory, thus making the probes stealthier.

Allocation oracles [54] scan the address space, indirectly. Rather than trying to access the allocated hidden regions, they infer their location by probing for unallocated holes in the address space. By trying many large allocations and observing whether they succeed or not, the attacker eventually finds the sizes of the random-sized holes, and hence the location(s) of the hidden regions.

Cache-based or similar timing side-channels form another class of derandomization primitives. AnC [34] for example, recently demonstrated using probes based on local cache accesses to bypass ASLR protection. Such attacks require access to the local system providing an execution environment for attackers' code (e.g., a Javascript engine). However, randomization-based defenses are primarily designed to protect against remote attacks and not against local attacks. A remote attacker targeting a server application cannot use such derandomization primitives because of lack of access to its local system.

Table 2 lists existing classes of derandomization primitives for a remote attacker, viz., arbitrary read, write and jump vulnerabilities along with memory allocation based primitive, and illustrates those suitable to attack the listed information hiding based modern defenses. For all these classes, ProbeGuard currently implements anomaly detectors and reactive hardening. Although this captures a wide set of foundational primitives, we do not claim the table to be exhaustive as researchers keep finding new primitives. What is important though, is that all derandomization techniques require multiple probing attempts before they eventually break the randomization. Since they must provide useful signals to the attacker, they all tend to have some unusual characteristics. As we shall see, ProbeGuard mitigates such attacks by reducing the number of probing attempts available to an attacker for a given primitive to just one detectable probe.

4 Overview

Applying ProbeGuard, after first protecting an application with any state-of-the-art information hiding-based defense (our baseline), ensures that the protected application is immune to derandomization. The resulting binary can then run in production. Figure 1 shows how ProbeGuard operates at runtime on a hardened application. An attacker may probe the application using any derandomization primitive in an attempt to break information hiding and bypass the baseline defense. Say, the attacker uses a buffer overflow vulnerability to corrupt a data pointer that the application reads from. In principle, she can use this arbitrary memory read primitive [24] to probe random memory addresses, looking for

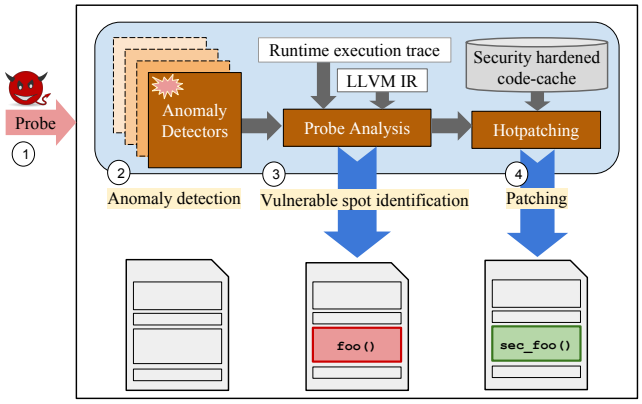

Figure 1. ProbeGuard's workflow: 1) An attacker makes a probing attempt; 2) One of the anomaly detector senses and triggers reactive hardening; 3) The probe analyzer identifies the offending spot; 4) The hotpatcher replaces the offending spot on-the-fly with its hardened variant.

the hidden region. However, a random probe most likely hits an invalid address in a huge 64-bit address space, triggering a segfault. ProbeGuard's *anomaly detection* detects this and triggers reactive hardening.

A detected anomalous event, temporarily stops the application and invokes *probe analysis*, which analyzes the current execution context to find the offending code fragment by utilizing the trace obtained from efficient and tamper-resistant branch tracing facilities available on modern processors (e.g., Intel PT). ProbeGuard lifts the trace (obtained via the kernel) by mapping binary instruction addresses back to its source information to precisely pinpoint the code fragment that the attacker used as a probing primitive —even under attack when we can no longer trust user memory.

Next, ProbeGuard's *hotpatching* component replaces just the pinpointed code fragment (function `foo()` in the figure) on the fly, with a semantically-equivalent but hardened version (function `sec_foo()` in figure 1). The new code fragment includes targeted integrity checks that stop the attacker's ability to use the offending primitive, at the cost of slowing down the execution of *just* that fragment. In the above example, ProbeGuard can insert software fault isolation (SFI) [65] checks in this code fragment, limiting the probe primitive's access to regions far away from the hidden region, thus protecting the hidden region from malicious accesses. ProbeGuard then activates the new code fragment by piggybacking on the recovery functionalities of the target application (which make these derandomization attacks possible in the first place), such as Nginx server forking to replace a crashed child. Further probing attempts using the same primitive, whether or not they lead to a crash, cease to produce desirable signals for the attacker.

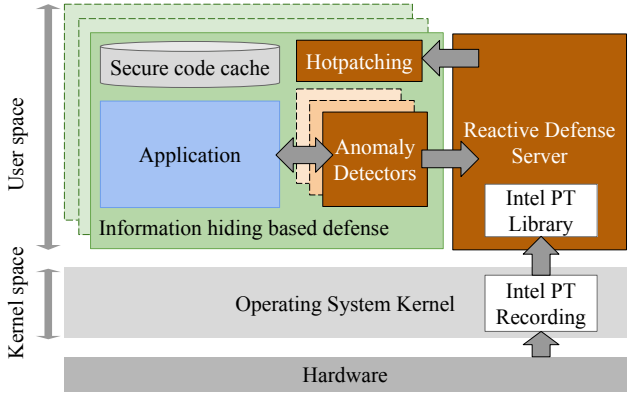

Figure 2. ProbeGuard architecture

5 Design

This section details the architecture and design of ProbeGuard. Our design goals are (i) to mitigate probing attempts on protected applications through reactive hardening, and (ii) to balance security and performance.

An application employs information-hiding based state-of-the-art defenses. ProbeGuard must ensure what is hidden remains hidden. Figure 2 shows the main components of ProbeGuard. We embed *anomaly detectors* within the application that sense probing attacks and a *code cache* consisting of a collection of code fragments hardened by applying LLVM [46]-based integrity checking instrumentations. A separate *reactive defense* server decodes execution traces obtained by Intel PT and performs fast *probe analyses*. ProbeGuard then reactively activates hardened code fragments by *hotpatching* when under attack. Following, we describe these components that make up ProbeGuard and discuss how they achieve our design goals.

We first discuss how we use anomaly detection to sense probing attacks. Next, we explain how ProbeGuard identifies the vulnerable code region that is under attack. Then we discuss how we apply hotpatching to add selective defenses to the affected region in the running application. Finally, we go through the various defenses that our prototype can apply for mitigating known attacks against information hiding.

5.1 Anomaly Detection

An attacker may use several classes of derandomization primitives. We employ dedicated anomaly detectors to efficiently and immediately detect any probing attempt.

Arbitrary reads and writes An attacker may exploit an arbitrary memory read or write vulnerability in the application with the goal of derandomizing the hidden region. Typically, only a very small fraction of the application's virtual address space is actually mapped. So, when the attacker uses such a vulnerability to access a random address, it is highly likely to hit an unmapped virtual memory address leading to a segmentation fault (or a crash). On UNIX-based

systems, for example, the operating system sends a SIGSEGV signal, typically resulting in an application crash (and automatic recovery, in case of our target applications). We detect such probing attacks by simply handling and proxying the signal using a custom SIGSEGV handler. Even in the case of buggy or unusual SIGSEGV-aware applications, this would not affect (or break) application behavior, but as a consequence, only increases the application's hardened surface.

Kernel reads and writes Attackers prefer probing silently and avoid detection. Hence, to avoid the crashes, they could also attempt to derandomize the victim application's address space by probing memory via the kernel. Certain system calls (e.g., read) accept memory addresses in their argument list and return specific error codes (e.g., EFAULT) if the argument is a pointer to an inaccessible or unallocated memory location. Using arbitrary-read/write primitives on such arguments, they could attempt CROP [25] attacks to enable probes eliminating application crashes (thereby not generating SIGSEGV signals). We can detect such probing attacks by intercepting system calls, either in glibc or directly in the kernel, and inspecting their results. As these events are, again, very unusual, we identify them as anomalies and trigger reactive hardening. In our prototype, we intercept the system calls at the library level, since doing so minimizes complexity in the kernel and benign applications that directly invoke system calls are extremely rare.

Arbitrary jumps Some vulnerabilities allow attackers to control the instruction pointer, effectively giving them an arbitrary jump primitive. For example, leakage-resilient techniques that defend against JIT-ROP attacks [59], such as XnR [11] and Readactor [19] are vulnerable to arbitrary jump primitives. However, these primitives may not help target other defenses—e.g., those that provide both forward- and backward-edge CFI protection. Arbitrary jump primitives allow scanning the address space looking for valid code pointers and then, locate code gadgets. BROP [14], for example, turns a stack write vulnerability into an arbitrary jump primitive. As in the case of arbitrary read and write vulnerabilities, an attempt to execute unmapped or non-executable memory results in either a segmentation fault (raising a SIGSEGV signal) or an illegal instruction exception (raising a SIGILL signal) as the memory region may not contain valid machine instructions. To detect these probing attacks, we extend our custom signal handler to handle both the signals and trigger reactive hardening as explained earlier.

Allocation oracles Oikonomopoulos et al. show that information hiding based defenses are susceptible to attacks that use allocation oracles [54]. Such probes exploit memory allocation functions in the target application by attempting to allocate large memory areas. Success or failure of the allocation leaks information about the size of holes in the address space, which in turn, helps locate the hidden region. We can detect these probes by looking for unusually large memory allocation attempts. We do so by hooking into glibc to intercept the system calls used to allocate memory (e.g., mmap() and brk()). The more widely used allocation library calls (e.g., malloc()) get intercepted indirectly as they internally rely on these system calls to obtain large memory areas from the operating system. We choose a configurable threshold on the allocation size, above which our detector triggers reactive hardening (half of the address space by default).

Other primitives While we covered all the widely used derandomization primitives, researchers may well find new primitives in the future. So, it is impossible to assure detection of all kinds of probes preemptively. Nonetheless, any probe must: (i) provide clear and distinct signals to the attacker—the same should help us in probe detection too, and (ii) probe *memory*; so, application-level detection will remain viable because a remote attacker has no access to other ways that use external or hardware-based side-channels as discussed earlier. We make ProbeGuard easily extensible to include new detectors whenever new primitives surface.

5.2 Probe Analysis

Upon an anomaly detector flagging a potential attack, ProbeGuard must determine the probing primitive used, or, in other words, locate the offending code fragment—which we refer to as "probe analysis". A derandomization primitive might as well make use of undetectable buffer over-read and over-write vulnerabilities that may write to some other pointers within a valid mapped memory area, which eventually get dereferenced elsewhere during the application's execution. We note that the final effect of the primitive (in this case, the spot where corrupted pointers are dereferenced) and its code location matters more than the location of the corresponding vulnerabilities, for inhibiting the attack. This is because it is the final manifestation of the vulnerability that gives the attacker the capability to derandomize the memory address space, which is what we refer to as a probing primitive. To locate the probing primitive, we employ hardware-assisted branch tracing to fetch the control flow prior to when we detected the anomaly. We build a reverse mapping to fetch source-level information from the trace. This is a key enabler for program transformation-based hotpatching in ProbeGuard.

We obtain past executed control-flow using Intel PT, which offers low-overhead and secure branch tracing. Control bits in the CPU's model-specific registers (MSRs) allow an operating system kernel to turn this hardware feature on or off. Intel PT stores highly compressed trace packets in a *circular buffer* in the kernel's memory space, beyond the reach of an attacker in user space. The buffer size is configurable; typical values range from 2 MB to 4 MB or more. ProbeGuard does not require a very deep peek into the past. We need the buffer to hold just enough to point beyond any execution

in library/external functions. (A similar execution tracing feature, Last Branch Record (LBR) from Intel saves the last 16 branches executed. However, this may stand insufficient to provide enough visibility into the past.) Although decoding the trace data is much slower than the fast recording, we rarely need to do this (i.e., upon being probed). We will show in Section 7.2 that even then, the processing times remain acceptable for our purposes, because the backward trace analysis can limit itself to the relevant recent control-flow history and avoid decoding all of the trace in its entirety.

On Linux, the `perf record` command interface allows users to trace Intel PT events on a per-process and even *per-thread* basis in the target application (using the `--per-thread` option). We use its snapshot mode [3, 40] and dump the trace when required; i.e., when an anomaly gets detected. Although the decoded trace provides the sequence of code addresses executed right until the detected anomaly, mapping them back to the source code and determining the offending code fragment is still challenging.

The *probe analyzer* must locate the affected spot in the source code. We repurpose a field in LLVM's debug metadata that normally carries `column number` of the source code location to instead place respective basic block identifiers. This only simplifies our prototype implementation to let LLVM's default code generator pass on the metadata through DWARF 4.0 symbols onto the resulting application binary, instead of having to use a new metadata stream and write the supporting code.

With this, we have a facility for reverse mapping from code addresses in the trace, onto the binary, all the way to where it belongs in the application's LLVM intermediate representation (LLVM IR or "bitcode"). Although ProbeGuard can identify the offending fragment at the basic block level, we choose to mark the entire parent function that includes the probing primitive and use this for hardening, as this strategy simplifies hotpatching and offers better security (see Section 7).

5.3 Hotpatching

Probe analysis provides the following information: (1) the particular code fragment under attack (the probing primitive), and (2) type of the derandomization primitive, as indicated by the anomaly detector that triggered the reactive hardening. Using these, ProbeGuard's hotpatcher can select appropriate security hardening to thwart any further probing attempts that use the same primitive.

To facilitate hotpatching, we first transform the program using our LLVM compiler passes. The goal is to be able to quickly and efficiently replace each vanilla variant of a function with a different (hardened) variant of the same function at runtime. We clone all functions found in the target application's LLVM IR and selectively invoke security-hardening instrumentation passes on specific function clones at *compile time*. The program executes the uninstrumented

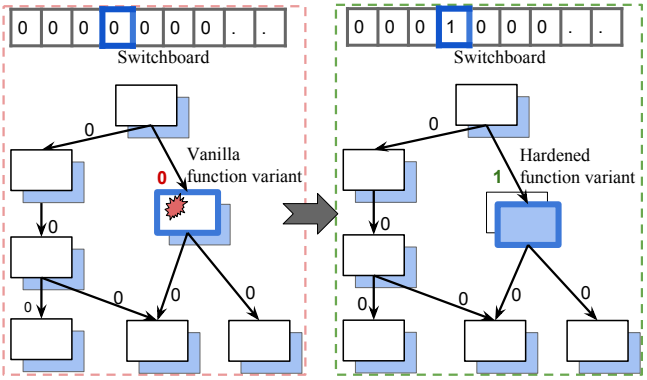

Figure 3. ProbeGuard's hotpatching strategy: call graph modifications in response to a probing attempt

variants by default, resulting in good performance, but has the set of instrumented variants available in a code cache to instantly switch to the appropriate instrumented variant at *runtime* when anomalous events demand better security.

Figure 3 depicts ProbeGuard's hotpatching strategy. A global `switchboard` (which we insert in the application) allows switching between each function variant at runtime. It contains an entry for each function in the program, controlling which of the variants to use during execution. We make every function in the application consult the switchboard and switch to its appropriate variant. In our current prototype we use only two variants: one for the vanilla version and one for the hardened version; the latter instrumented with all the supported hardening techniques. While we can easily support more variants and patch each affected function with the variant hardened against the offending primitive type, the current design is simpler, provides better memory usage, and better performance during regular execution (but worse during hardened variant execution).

To deter attacks against ProbeGuard, we mark the switchboard as read-only during normal execution. We can also rely on information hiding itself to protect the `switchboard` as done for our hardening techniques as necessary, given that ProbeGuard can already stop all the probing attacks against arbitrary hidden regions.

5.4 Selective security hardening

Having seen all of probe detection, probe analysis and hotpatching in ProbeGuard, we now look at the instrumentations we use for reactive hardening –a set covering all the fundamental probe-inhibiting integrity defenses: limiting read and write accesses, setting thresholds on data values and preventing targeted control-flow diversions. Thwarting a probing primitive implies stopping it from producing a usable signal for derandomization. For example, a probing primitive, when hotpatched produces crashes for any illegitimate memory access—whether within mapped or unmapped memory areas. So, the primitive no longer remains usable

for probing as it ceases to provide perceivable signals to the attacker. We base our selection of defenses to apply for each attack on the options presented in Table 2.

Arbitrary reads and writes Software Fault Isolation (SFI) mitigates probing attempts that use arbitrary reads and writes. It simply instruments every load or store operation in the application binary by masking the target memory location with a bitmask. For example, in our prototype, within the usable 48 bits of 64-bit virtual address space, we ensure that the 47^{th} bit of the memory pointer used within the target application is always *zero* before dereferencing it (only the deployed code reuse defense instrumentations continue to access the hidden region as they should). Thus, by restricting the hidden region to virtual addresses with the 47^{th} bit set (hidden address space), the attacker can no longer use an SFI-instrumented function for probing. Although we lose one bit of entropy, this makes it much more secure by protecting the remaining bits.

Kernel reads and writes While we cannot reactively apply SFI within the kernel itself, we can apply a variation in the application to defend against kernel-based reads and writes. We mask all pointer arguments to library calls in the same way we mask loads and stores against arbitrary reads and writes. This ensures that the attacker cannot perform system calls that access hidden regions. The checks take into account any size arguments that may otherwise help in bypassing the defense.

Arbitrary jumps Targeted CFI checks can mitigate arbitrary jump primitives. CFI restricts the program to its known and intended sets of control flow transfers [8]. Its strictest form is rarely used in practice as it incurs a significant performance overhead. Numerous CFI variants in the past have sought to balance security and performance, but studies [30] show that toning down security guarantees by any margin exposes CFI to practical attacks. However, our goal is not to protect the entire application from code reuse attacks (the baseline defense does that already), but to prevent the attacker from using the same probing primitive again to reveal the hidden regions. For this purpose, we can use even the strongest CFI protection without much overhead. In our current prototype, we implement the following checks to neutralize probes that divert control flow.

Forward-edge protection: An attacker can corrupt a code pointer used by a particular indirect call instruction for probing purposes. We can prevent this attack if we label every potential target of an indirect call (address of any function that has its address taken) and instrument indirect calls to verify that the call target has a matching label. We can use static analysis at compile-time to determine which labels are potential targets for each indirect call. The more restrictive the set of possible target labels, the better the CFI protection. As our focus is more on evaluating the overall impact of selective hardening, we implemented a type-based CFI policy similar to IFCC [61] in our current prototype. However, in a selective hardening scenario, more sophisticated policies, normally inefficient at full coverage (e.g., context-sensitive CFI [63] piggybacking on the full Intel PT traces available in ProbeGuard), are also viable.

Backward-edge protection: Alternatively, an attacker could corrupt return addresses on the stack to divert control flow and probe the application's address space. We implement a per-thread shadow stack that stores return addresses to be able to prevent such control-flow diversions. We statically instrument function entry points to push the return address onto the shadow stack and at function return points to check that the return address is still the same as the one in the shadow stack. We protect the shadow stack itself using information hiding by randomly placing it in the hidden address space. We prevent any attempt to detect its location by reactively deploying our other defenses (e.g., SFI) as necessary. Targeted function-wise protection by shadow stack suffices against probes because, without a detectable probing attempt elsewhere in the code base, an attacker cannot influence unprotected parts of the call stack, particularly for reconnaissance.

Allocation oracles To mitigate probing attacks that aim to perform memory scanning through memory allocation primitives, we apply a threshold on the size arguments of library functions that provide memory allocation utilities, such as the `malloc` family of functions by instrumenting their call sites. Though, we note that applications may perform very large allocations during their initialization phase. A completely agnostic threshold-based anomaly detector would prevent even such legitimate memory allocations. We use a white-listing scheme for such cases, distinguishing them by the nature of the size argument. If this argument originates from a constant in the application (i.e., a value the attacker cannot control by construction), or even defenses like CPI [44]– which initially reserves huge constant-sized buffers for shadow memory-based metadata management, we deem them to be harmless.

6 Implementation

Module	*Type*	*#SLOC*
Anomaly detection	C static library	598
	Changes to `glibc`	51
Reactive Defense Server	Python	178
Probe Analysis	C program	1,352
Hotpatching	C++ LLVM passes	1,107
Hardening	C static libraries	340
	C++ LLVM passes	1,332

Table 3. SLOC counts of ProbeGuard's modules.

ProbeGuard's implementation consists of the following:

1. A static library linked with the application: It houses a signal handler registered at startup. The signal handler takes actions depending on the type of anomaly; It also interposes on application-defined signal handler registrations (e.g., `sigaction` calls) to preserve and chain invocations. Finally it helps in hotpatching to support switching between function variants at runtime.
2. `glibc` modifications to intercept `mmap()`-like syscalls to detect huge allocation primitives and syscalls that result in `EFAULT` to detect CROP-like primitives.
3. LLVM compiler passes to generate and propagate function identifying markers onto the binary via DWARF 4.0 symbols (necessary to build reverse mappings) and function cloning to facilitate hotpatching.
4. A separate reactive defense server that does probe analysis by fetching Intel PT traces using `libipt` [4] to map them onto the binary by reading the markers using `libdwarf` [7].

Besides these, we implemented other LLVM instrumentation passes for hardening that insert SFI, CFI, and allocation-size checks selectively at function granularity.

Table 3 shows the number of source lines of code (SLOC) that we wrote to implement ProbeGuard, as reported by `SLOCCount`. Our anomaly detection components interact with the reactive defense server via traditional inter-process communication (i.e., UNIX domain sockets). This is to request probe analysis and receive the result. Based on the result, we perform hotpatching by updating the global switchboard that switches the offending code fragment with its corresponding hardened variant.

In principle, a binary-only implementation of ProbeGuard is also possible. Our probe analysis already maps code locations in Intel PT trace dump to their counterparts in the binary using DWARF 4.0 based markers (we even extend it to LLVM IR). Binary rewriting techniques can support implementing a global switchboard based control of function variants. We chose a source-level implementation because many information hiding based defenses we aim to protect also happen to rely on source code based analysis and transformation techniques.

7 Evaluation

We evaluated our ProbeGuard prototype on an Intel i7-6700K machine with 4 CPU cores at 4.00 GHz and 16 GB of DDR4 memory, running the 64-bit Ubuntu 16.04 LTS Linux distribution. We compared programs instrumented by ProbeGuard against a baseline without any instrumentation. We use an uninstrumented baseline to simulate a configuration akin to an ideal information hiding-based defense (and thus as efficient as possible). We note that this is a realistic setup, as many information hiding-based defenses report performance figures which are close to this ideal baseline. For example, Safe-stack reports barely any overhead at all in standard benchmarks [44]. Our instrumented version, on the other hand, supports all the integrity-based defenses detailed in the paper and combines all of them together into a single hardened variant for each function in the program.

We evaluated ProbeGuard on the SPEC CPU2006 benchmarks as well as on the Nginx web server, which has been repeatedly targeted by probing attacks. To benchmark the web server, we used ApacheBench [6], issuing 25,000 requests with 10 concurrent connections and 10 requests per connection, sufficient to saturate the server. Our set of programs, benchmarks, and configurations reflect choices previously adopted in the literature.

Our evaluation focuses on five key aspects of ProbeGuard: (i) *performance overhead* (during regular execution, how fast is a ProbeGuard-instrumented version of a program?), (ii) *service disruption* (what is the impact on the execution during repeated probing attack attempts, each triggering trace decoding and hotpatching?), (iii) *memory overhead* (how much more memory does a ProbeGuard-instrumented version of a program use?), (iv) *security* (what is the residual attack surface?), (v) *effectiveness* (can ProbeGuard stop existing probing-based exploits?).

7.1 Performance overhead

We first evaluated the overhead that ProbeGuard alone adds during regular (attack-free) execution, on the full set of SPEC CPU2006 benchmarks. This measures the overhead of our runtime components along with Intel PT branch tracing. As shown in Figure 4, the average (geomean) overhead of our solution is only 1.4%. Figure 4 also shows the normalized performance overhead of the individual integrity defenses when applied throughout the application during regular execution—SFI, CFI (both forward and backward edge protection) and AllocGuard (allocation-size thresholding), with average (geomean) overheads of such defenses being 22.9%, 11.5% and 1.3% respectively, along with an all-combined variant with an overhead of 47.9%, which is much higher than our solution. This stems from ProbeGuard's basic instrumentation being lightweight, with essentially a zero-overhead anomaly detection. The residual overhead stems from Intel PT's branch tracing activity (which can also be used to support other defenses) and slightly worse instruction cache efficiency due to larger function prologues (padded with a `NOP` sled). The latter overhead is more prominent in benchmarks that contain very frequent function calls in the critical path (e.g., lbm, povray and perlbench).

Further, we measured throughput degradation in Nginx server by running the Apache benchmark. The attack-free ProbeGuard-instrumented version of the server reported a degradation of only 2.4% against the baseline. This demonstrates that ProbeGuard is effective in significantly reducing the overhead of full-coverage integrity-based solutions, while retaining most of their security benefits.

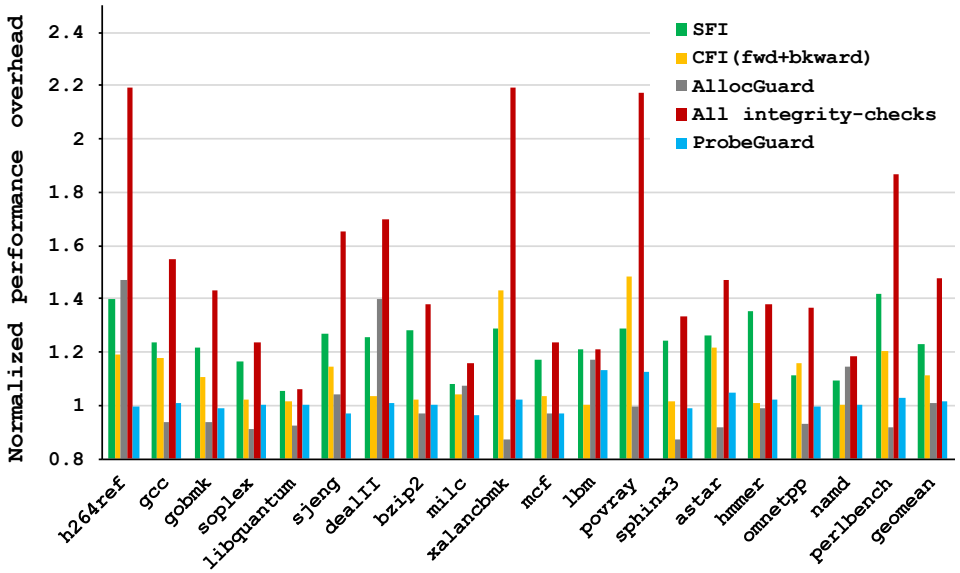

Figure 4. SPEC CPU2006: ProbeGuard (1.4%) vs. full-coverage integrity defenses (47.9%) during regular execution.

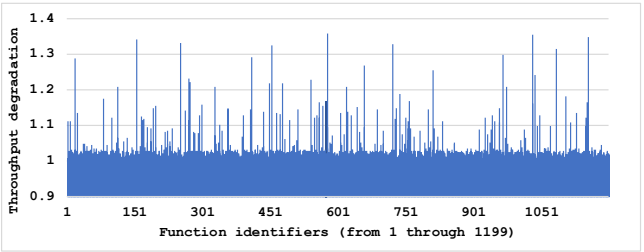

Figure 5. Performance impact of hardening each Nginx's function separately.

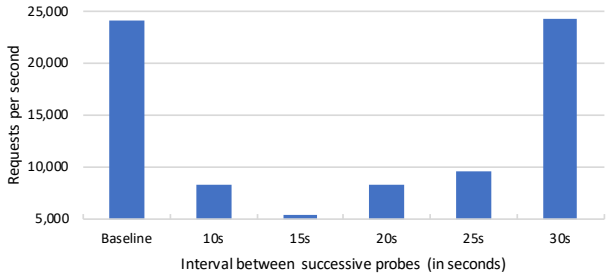

Figure 6. Throughput degradation observed on Nginx for varying probing intervals

In order to assess how overhead varies when an ideal attacker locates several probing primitives, we measured the overhead *separately*, that *each* function adds upon hardening, in Nginx, shown in figure 5. It shows that frequently executed functions have greater impact and as we see, the *worst-case* function (i.e., on the critical path) has an impact of 36% on the throughput. However, in practice, bugs are less likely to remain in critical paths of well-tested applications.

7.2 Service disruption

To simulate worst-case attack conditions, we also subjected the ProbeGuard-instrumented Nginx server to repetitive probing attempts, in increasing intervals. Although, in practice, a heavy influx of probing attacks is highly unlikely, given that it would require uncovering a huge number of *unique* probing primitives (each in a distinct function), this serves as a stress benchmark for on-the-fly probe analysis and hotpatching that piggybacks on the server's inherent crash recovery functionalities (throughout which the server remains temporarily frozen). Figure 6 depicts the throughput degradation incurred by the Nginx web server for varying probing intervals. For probing intervals of 10, 15 and up to 25 seconds, throughput drops between 60% - 78%. We can attribute the degradation to crashes induced by successive probing attempts. However, with larger intervals between the probes viz., 30 seconds onward, we saw no observable impact on the throughput. This clearly shows that probe analysis and hotpatching do not adversely affect service availability even under aggressive attacks (even though such attack rates are infeasible in practice).

7.3 Memory overhead

We measured the memory overhead of ProbeGuard on the SPEC CPU2006 benchmarks. The computed resident set size (RSS) remains marginal (1.2% on average, geometric mean) during regular execution. On Nginx, while running the same Apache benchmark, we saw a mean increase in RSS memory usage of approximately 350KB, which would include a constant size additionally occupied by the switchboard. This shows that ProbeGuard can be realistically applied to real-world applications with low memory overhead.

7.4 Security

We evaluate ProbeGuard's security guarantees against concerted probing attacks on information hiding-based defenses and then discuss potential strategies for an attacker to circumvent ProbeGuard.

A probing attack follows a strike-and-observe pattern, typically involving several attempts before leaking precious hidden sensitive information from the victim application. Table 4 depicts the security guarantees that ProbeGuard offers for a number of representative hidden region sizes drawn from common information hiding-based defenses (using Nginx as a reference). As shown, such sizes may range from an ideal case of a single memory page (4 KB) to the few GBs of virtual memory CPI uses, with their entropy respectively ranging from 34 to 14 bits. Note that we calculated the entropy for CPI's hashtable and lookup table (CPI's recommended information hiding configurations) based on the conservative estimates reported by the authors for a 1 GB resident set size (RSS) [45].

Compared to traditional information hiding, ProbeGuard sacrifices one bit of entropy (SFI enforcement) starting from a 47-bit user-addressable memory address space. This doubles the probability $P(region|\#p = 1)$ of locating the hidden region at the first attempt with a given primitive (except for allocation primitives, where ProbeGuard preemptively stops any probing attempts in any case). Nonetheless, such probability remains extremely low (2^{-14} in the worst case—CPI's lookup table) to mount a practical attack. Moreover, after the first attempt, traditional information hiding imposes no restriction on the attacker, whereas ProbeGuard triggers reactive hardening that stops any further use of the same probing primitive. The only option the attacker has is to locate any other primitive in a different function to try probing again. For example, an attacker can use arbitrary read or write primitives to follow pointers and traverse all valid data memory. Thereafter, she can modify any pointer that will be dereferenced along any execution path (possibly in several different functions) that can be invoked remotely to probe the memory. While this is already impractical as this requires uncovering several probing primitives, the underlying entropy dictates locating as many as $2^{14} = 16,384$ primitives, *each in a distinct function* in the best case (CPI's lookup table) for the attack to be fully reliable. This is hard in general and for an Nginx-like server program in particular, given that it only contains 1,199 functions in total. Even in the ideal, non-realistic scenario where an attacker has found one primitive for each function in the program, the probability of locating the hidden region $P(region|\#p = MAX)$ is still insufficient to mount practical attacks for all the hidden region sizes considered. Nonetheless, Table 4 does show that the hidden region size has a strong impact on the security upper bound guaranteed by ProbeGuard.

We now consider other strategies an attacker may employ to attack ProbeGuard. First, an attacker may attempt to craft new, unknown probing primitives not yet supported by ProbeGuard. While this is a possibility, ProbeGuard is also extensible to support detection of new primitives. Nevertheless, we cannot discount the possibility of new primitives that would be hard to adequately support in the existing framework (e.g., if anomaly detection cannot be easily implemented in a lightweight, low-overhead fashion). Note, however, that ProbeGuard currently covers support for all sets of fundamental primitives and many new primitives may ultimately resort to using these existing ones to mount end-to-end attacks. For example, our current prototype cannot detect thread spraying primitives [31] (although we can extend it to do so). However, an end-to-end thread spraying attack still requires an arbitrary memory read/write probing primitive, which ProbeGuard can detect and hotpatch.

Second, an attacker may try to locate primitives in as many functions as possible, not necessarily to reveal the hidden region, but to intentionally slow down a victim application. While this is theoretically possible, we expect the number of primitives (usable primitives in distinct functions) in real-world applications to be sufficiently limited to deter such attacks. Similarly, one can mount surface expansion attacks, for example if the attacker learns that one of our reactive hardening techniques has an implementation bug. She could lure ProbeGuard to hotpatch some function that injects a previously non-existent vulnerability into the application. More generally, an attacker could target implementation bugs in the baseline defense or our infrastructure to bypass ProbeGuard. While we cannot discount the possibility of such bugs in baseline defenses, ProbeGuard itself has a relatively small trusted computing base (TCB) of around 5,000 SLOC to minimize the attack surface.

Finally, an attacker may circumvent the code reuse defense without derandomizing and revealing hidden sensitive data. For example, using arbitrary read/write primitives, an attacker could conservatively walk through memory without touching unmapped memory and avoid detection. Even though this restricts such probes to regular non-hidden memory regions of the application, an attacker may choose to exploit memory disclosures to target defenses against JIT ROP [59] attacks for example, that build and rely on leakage resilience [11, 13, 16, 17, 19, 66]. We focus on hardening arbitrary code reuse defenses against information hiding attacks which have shown to trivially bypass even advanced defenses. We make no attempt to address other design weaknesses of such defenses, such as leakage-resistant code randomization being vulnerable to sophisticated code-reuse attacks [32, 56, 64].

7.5 Effectiveness

We tested our prototype's effectiveness in stopping all existing probing-based exploits against information hiding,

Hidden region	*Size*	*Entropy (bits)*	*P(region|#p=1)*	*P(region|#p=MAX)*
Ideal (one memory page)	4 KB	(47 - 1) - 12.0 = 34.0	$2^{-34.0}$	$2^{-23.8}$
Shadow stack (single thread)	8 MB	(47 - 1) - 23.0 = 23.0	$2^{-23.0}$	$2^{-12.8}$
Shadow stack (32 threads)	256 MB	(47 - 1) - 28.0 = 18.0	$2^{-18.0}$	$2^{-07.8}$
CPI's hashtable (1 GB RSS)	1.4 GB	(47 - 1) - 30.4 = 15.6	$2^{-15.6}$	$2^{-05.4}$
CPI's lookup table (1 GB RSS)	4 GB	(47 - 1) - 32.0 = 14.0	$2^{-14.0}$	$2^{-03.8}$

Table 4. ProbeGuard's security guarantees (Nginx used as a reference, with 1,199 functions). $P(region|\#p = k)$ is the probability of locating the hidden region once the attacker discovers primitives in k different functions in the program.

viz., Blind ROP (BROP) [14], remote arbitrary memory read-/write primitives [57], server-side Crash-Resistant Oriented Programming (CROP) [41], and allocation oracles [54].

To evaluate ProbeGuard's effectiveness in stopping BROP (arbitrary jump) probing attacks, we downloaded and ran the BROP exploit [2]. It repetitively uses a stack-based buffer overflow in the function `ngx_http_parse_chunked` in nginx 1.4.0 (CVE-2013-2028) to corrupt the return address and divert control flow upon function return to probe its address space based on crash or no-crash signals. Without ProbeGuard, the exploit ran successfully. With ProbeGuard, the exploit no longer succeeded: at the first (failed) jump-based probing attempt, ProbeGuard detected the event and reactively hardened (only) the offending function with a shadow stack. All subsequent control-flow diversion attempts through this function invariably resulted in crashes, thwarting the probing primitive as it could no longer produce any useful signal for the attacker.

To evaluate ProbeGuard's effectiveness in stopping arbitrary memory read/write-based probing primitives, we reproduced a stack-based buffer overflow vulnerability in the `sreplace()` function in proftpd 1.3.0 (CVE-2006-5815), using the publicly available exploit [1]. By controlling the arguments on the stack, an attacker can use a call to `sstrncpy()` to write to arbitrary memory locations [37]. Without ProbeGuard, the attack could probe the address space for mapped (writable) memory regions and locate a sensitive target. With ProbeGuard, the first such write to an unmapped memory area triggered reactive hardening of the offending function with SFI. This indiscriminately prevented all the subsequent arbitrary memory write attempts, effectively thwarting this probing primitive.

To evaluate whether ProbeGuard can stop CROP (kernel memory read/write) probing attacks, we used such an attack described by Kollenda et al. [41]. Locating the next client connection via `ngx_cycle->free_connections` before sending a partial HTTP GET request, the attacker exploits a kernel memory write primitive to probe a chosen memory region by controlling the connection buffer (`ngx_buf_t`) parameters. If the chosen region is neither mapped nor writable memory, the `recv()` system call returns an `EFAULT`, forcing the server to close the connection. Otherwise, if the chosen memory was writable, the server successfully returns the requested page. Without ProbeGuard, the attack completed successfully. With ProbeGuard, our glibc `EFAULT` interceptors detected an anomalous event, reactively hardening (only) the offending function with SFI. The latter indiscriminately prevented all the subsequent kernel memory write attempts through this function, thwarting this probing primitive.

To evaluate ProbeGuard against allocation oracles attacks, we downloaded and ran the publicly available exploit [5] on Nginx 1.9.6 (the version on which the attack was originally tested). Without ProbeGuard, the exploit successfully derandomized the address space, revealing the sensitive memory region. With ProbeGuard, even the first probe failed as our interceptors in glibc enforced allocation size thresholds and triggered reactive hardening.

8 Conclusion

Many researchers today believe that defenses based on randomization are doomed and more heavy-weight solutions are necessary. This paper showed how reactive defenses can bring together the best of both worlds and transition from inexpensive passive defenses to stronger but expensive active defenses when under attack, incurring low overhead in the normal case, while approximating the security guarantees of powerful active defenses. Our evaluation showed that such a solution for generic Linux programs is effective at balancing performance and security. To foster more research in the area, we open source our ProbeGuard prototype[1] as a general framework to build future reactive defenses.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct), by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by Cisco Systems, Inc. through grant #1138109 and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI "Dowsing" and grant NWO 639.021.753 VENI "PantaRhei". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

[1] https://github.com/vusec/probeguard

References

[1] 2006. Proftpd CVE 2006-5815. https://www.exploit-db.com/exploits/2856/. (2006).

[2] 2014. BROP Nginx exploit. http://www.scs.stanford.edu/brop/nginx-1.4.0-exp.tgz. (2014).

[3] 2014. perf: Add infrastructure and support for Intel PT. https://lwn.net/Articles/609010/. (2014).

[4] 2015. Intel Processor Trace decoder library. https://github.com/01org/processor-trace. (2015).

[5] 2016. Poking Holes. https://github.com/vusec/poking-holes. (2016).

[6] 2018. ApacheBench. (2018). http://httpd.apache.org/docs/2.4/programs/ab.html

[7] 2018. 'libdwarf' library. https://www.prevanders.net/dwarf.html. (2018).

[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *CCS*.

[9] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *TISSEC* 13, 1 (2009).

[10] Jeff Arnold and M Frans Kaashoek. 2009. Ksplice: Automatic rebootless kernel updates. In *EuroSys*.

[11] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *CCS*.

[12] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security*.

[13] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *CCS*.

[14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *S&P*.

[15] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.

[16] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices.. In *NDSS*.

[17] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *EuroS&P*.

[18] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*.

[19] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*. IEEE.

[20] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ASIACCS*.

[21] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *ACM DAC*.

[22] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming.. In *NDSS*.

[23] Gregory J Duck and Roland HC Yap. 2016. Heap bounds protection with low fat pointers. In *ACM CC*.

[24] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point (er): On the effectiveness of code pointer integrity. In *S&P*.

[25] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*.

[26] Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *LISA*.

[27] Cristiano Giuffrida, Calin Iorgulescu, and Andrew S. Tanenbaum. 2014. Mutable Checkpoint-Restart: Automating Live Update for Generic Server Programs. In *Middleware*.

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. Safe and Automatic Live Update for Operating Systems. In *ASPLOS*.

[29] C. Giuffrida and Andrew S. Tanenbaum. 2012. Safe and Automated State Transfer for Secure and Reliable Live Update. In *HotSwUp*.

[30] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *S&P*.

[31] E Göktas, R Gawlik, B Kollenda, E Athanasopoulos, G Portokalidis, C Giuffrida, and H Bos. 2016. Undermining information hiding (and what to do about it). In *USENIX Security*.

[32] Enes Goktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. 2018. Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *EuroS&P*.

[33] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[34] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. *NDSS* (2017).

[35] Istvan Haller, Jeon Yuseok, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *CCS*.

[36] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks, and Jeffrey S Foster. 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. In *OOPSLA*.

[37] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*. IEEE.

[38] Intel. Processor Tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. (????).

[39] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM CCS*.

[40] Andi Kleen. https://lwn.net/Articles/648154/. (????).

[41] Benjamin Kollenda, Enes Göktas, Tim Blazytko, Philipp Koppe, Robert Gawlik, Radhesh Krishnan Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards Automated Discovery of Crash-Resistant Primitives in Binaries. In *DSN*.

[42] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys*.

[43] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks Without the Checks. In *EuroSys*.

[44] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer integrity. In *OSDI*.

[45] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, and Dawn Song. 2015. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *S&P*.

[46] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[47] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *ACM CCS*.

[48] Kristis Makris and Rida A Bazzi. 2009. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*.

[49] Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, and Dan Boneh. 2015. Cryptographically enforced control flow integrity. In *ACM CCS*.
[50] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *NDSS*.
[51] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *PLDI* (2009).
[52] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*.
[53] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical Dynamic Software Updating for C. In *PLDI*.
[54] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *USENIX Security*.
[55] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.
[56] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, and others. 2017. Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. (2017).
[57] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *CCS*.
[58] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*.
[59] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P*. IEEE.
[60] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*.
[61] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*.
[62] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *EuroSys*.
[63] Victor van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *CCS*.
[64] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *CCS*.
[65] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *SOSP*.
[66] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*.
[67] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *S&P*.
[68] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.