Undermining Information Hiding (And What to do About it)

Enes Göktaş[1] Robert Gawlik[2] Benjamin Kollenda[2] Elias Athanasopoulos[1]
Georgios Portokalidis[3] Cristiano Giuffrida[1] Herbert Bos[1]
[1] *Computer Science Institute, Vrije Universiteit Amsterdam, The Netherlands*
[2] *Horst Görtz Institut for IT-Security (HGI), Ruhr-Universität Bochum, Germany*
[3] *Department of Computer Science, Stevens Institute of Technology*

Abstract

In the absence of hardware-supported segmentation, many state-of-the-art defenses resort to "hiding" sensitive information at a random location in a very large address space. This paper argues that information hiding is a weak isolation model and shows that attackers can find hidden information, such as CPI's SafeStacks, in seconds—by means of *thread spraying*. Thread spraying is a novel attack technique which forces the victim program to allocate many hidden areas. As a result, the attacker has a much better chance to locate these areas and compromise the defense. We demonstrate the technique by means of attacks on Firefox, Chrome, and MySQL. In addition, we found that it is hard to remove all sensitive information (such as pointers to the hidden region) from a program and show how residual sensitive information allows attackers to bypass defenses completely.

We also show how we can harden information hiding techniques by means of an Authenticating Page Mapper (APM) which builds on a user-level page-fault handler to authenticate arbitrary memory reads/writes in the virtual address space. APM bootstraps protected applications with a minimum-sized safe area. Every time the program accesses this area, APM authenticates the access operation, and, if legitimate, expands the area on demand. We demonstrate that APM hardens information hiding significantly while increasing the overhead, on average, 0.3% on baseline SPEC CPU 2006, 0.0% on SPEC with SafeStack and 1.4% on SPEC with CPI.

1 Introduction

Despite years of study, memory corruption vulnerabilities still lead to control-flow hijacking attacks today. Modern attacks employ code-reuse techniques [9, 34] to overcome broadly deployed defenses, like data-execution prevention (DEP) [5] and address-space layout randomization (ASLR) [30]. Such attacks are still possible primarily because of address leaks, which are used to discover the location of useful instruction sequences, called gadgets, that can be chained together to perform arbitrary computations [34].

In response, researchers have been exploring various directions to put an end to such attacks. A promising solution is code-pointer integrity (CPI) [24] that aims to prevent the hijacking of code pointers, and therefore taking control of the program. The separation of code pointers from everything else can be done by employing hardware or software-enforced isolation [39,42], or by *hiding* the region where pointers are stored, which is a faster alternative, than software-based isolation, when hardware-based isolation is not available. This *information hiding (IH)* is achieved by placing the area where code pointers are stored at a random offset in memory and ensuring that the pointer to that area cannot be leaked (e.g., by storing it in a register). For example, safe versions of the stack, referred to as safe stacks, that only include return addresses are protected this way both by CPI and ASLR-guard [26]. This type of IH is also adopted by other defenses [7, 15] that aim to prevent attacks by eliminating data leaks, which would enable the location of gadgets, while it has also been adopted in shadow stack [13, 41] and CFI [27, 43] research.

Reliance on information hiding is, however, problematic. Recently published work [18] developed a memory scanning technique for client applications that can survive crashes. It exploits the fact that browsers, including Internet Explorer 11 and Mozilla Firefox, tolerate faults that are otherwise critical, hence, enabling memory scanning to locate "hidden" memory areas. Before that researchers demonstrated that it was possible to locate CPI's safe region, where pointers are stored [17], if IH is used instead of isolation.

In this paper, we reveal two new ways for defeating information hiding, which can be used to expose the "hidden" critical areas used by various defenses and subvert them. The first is technique caters to multithreaded applications, which an attacker can cause a process to spawn

multiple threads. Such applications include browsers that now support threads in Javascript and server applications that use them to handle client connections. By causing an application to spawn multiple threads, the attacker "sprays" memory with an equal number of stacks and safe stacks. As the address space fills with these stacks, the probability of "striking gold" when scanning memory increases dramatically. We incorporate this technique, which we coin *thread spraying*, in the memory scanning attack described above [18] and show that we can locate safe regions, such as the safe stacks used by CPI and ASLR-guard and parallel shadow stacks [14], in *seconds* instead of tens of minutes. The second approach utilizes bookkeeping data of various standard libraries in Linux such as the POSIX threads library and glibc. Our investigation reveals several pointers that can lead to safe regions in information kept to manage thread local storage (TLS) and thread-control blocks (TCB). Isolating these leftover pointers with a better implementation might be possible. However, at the time of writing, there is no algorithm for assessing if *all sensitive* pointers are properly removed. We therefore argue that a sound implementation which excludes all pointers that can lead to a safe region from the rest of the process is challenging.

These two new attack vectors demonstrate that it is surprisingly hard to use information hiding to replace strong isolation. Nevertheless, it is possible to further harden IH and mitigate these attacks. We propose and develop a technique that aims to shift the odds back in favor of the defender. Our solution is based on two observations: first, only a small part of the memory allocated to safe areas is actually used, and, second, safe areas are usually accessed using well-defined instrumentation-induced instruction patterns that are distinguishable from application code. For example, under CPI only a small part of the safe region and stacks allocated by CPU is actually used, and, on x86-64 architectures, the `gs` segment register and the `RSP` register are used to access the safe region and safe stacks, respectively.

Building on these observations, we introduce APM (Authenticating Page Mapper), an IH hardening strategy that verifies that all the first accesses to inactive pages in safe areas follow legitimate and unforgeable execution patterns, increasing in this fashion the entropy of IH. APM is implemented as a user-level page-fault handler, exploiting the same page "trapping" technique used by mechanisms like checkpoint/restart [31], hot patching [4], and ATL+DEP [1]. To further improve security, APM artificially inflates the virtual memory region reserved for safe areas. In particular, it decreases the number of accessible pages on a shadow stack to the minimum, but reserves many (initially inaccessible) additional pages that serve as tripwire. All illegitimate accesses to non-accessible pages trigger an alarm. Likewise, we offset the stack pointer by a random number of bytes.

In summary, we make the following contributions:

1. We introduce a new attack vector, called thread spraying, that reduces the entropy of information hiding techniques by creating many threads and filling up the memory space with safe areas such as safe stacks. We demonstrate the practicality of thread spraying for browsers (Firefox and Chrome) and the MySQL server, all protected with CPI's SafeStack, and show that we can find the hidden stacks in seconds.
2. We analyze in detail the strength of information hiding techniques used in state-of-the-art defenses and show that existing solutions are not free of pointers to safe areas. Our results enable an adversary to utilize these pointers to compromise the safe area.
3. We propose a new IH hardening strategy to improve information hiding by means of a user-level page fault handler that increases the entropy significantly, makes pages accessible on demand, and vets all first-time accesses to pages—greatly increasing the probability of detecting an attack. We evaluate the solution using a variety of applications as well as the SPEC benchmarks and show that the performance impact is very low (on average 0.3% on baseline SPEC, 0.0% on SPEC with SafeStack, 1.4% on SPEC with full CPI and barely measurable in browser benchmarks).

2 Threat Model

In this paper, we assume a determined attacker that aims at exploiting a software vulnerability in a program that is protected with state-of-the-art defenses (e.g., CPI [24] or ASLR-Guard [26]), and that has the capabilities for launching state-of-the-art code-reuse attacks [11, 16, 20]. We also assume that the attacker has a strong primitive, such as an arbitrary read and write, but the arbitrary read should *not* be able to reveal the location of a code pointer that could be overwritten and give control to the attacker, *unless* the safe area is somehow discovered. Under this threat model, we discuss in Sections 3 and 4 a number of possible strategies that can leak the safe area to the attacker. Later in this paper, we propose to harden IH using a technique that can effectively protect the safe area with a small and practical overhead.

3 Background and Related Work

In the following, we review relevant work on information hiding. We discuss both attacks and defenses to provide an overview of related work and hint at potential weaknesses. We show that prior work has already bypassed

several IH approaches, but these attacks all targeted defenses that hide very large areas (such as the 2^{42} byte safe area in CPI [17], or all kernel memory [22]). It is a common belief that smaller regions such as shadow stacks are not vulnerable to such attacks [26]. Later, we show that this belief is not always true.

3.1 Information Hiding

Many new defenses thwart advanced attacks by separating code pointers from everything else in memory. Although the specifics of the defenses vary, they all share a common principle: they must prevent malicious inputs from influencing the code pointers (e.g., return addresses, function pointers, and VTable pointers). For this reason, they isolate these pointers from the protected program in a *safe area* that only legitimate code can access in a strictly controlled fashion. In principle, software-based fault isolation (SFI [39]) is ideal for applying this separation. However, without hardware support, SFI still incurs nontrivial performance overhead and many defenses therefore opted for (IH) as an alternative to SFI. The assumption is that the virtual address space is large enough to *hide* the safe area by placing it in a random memory location. Since there is no pointer from the protected program referencing explicitly the safe area, even powerful information disclosure bugs [35] are useless. In other words, an attacker could potentially leak the entire layout of the protected process *but not* the safe area.

In recent years, this topic received a lot of attention and many systems emerged that rely (at least optionally) on IH. For example, Opaque CFI [27] uses IH for protecting the so called *Bounds Lookup Table* (BLT) and Oxymoron [7] uses IH for protecting the *Randomization-agnostic Translation Table* (RaTTle). Isomeron [15] needs to keep the *execution diversifier data* secret while StackArmor [41] isolates particular (potentially vulnerable) stack frames. Finally, CFCI [44] needs to hide, when segmentation is not available, a few MBs of protected memory. Although all these systems rely on IH for a different purpose, they are vulnerable to memory scanning attacks which try to locate these regions in a brute-force manner (as shown in Section 4). Since the Authenticating Page Mapper that we propose in this paper hardens IH in general, it directly improves the security of all these systems—irrespective of their actual goal.

3.2 ASLR and Information Leaks

Arguably the best known IH technique is regular *Address Space Layout Randomization* (ASLR). Coarse-grained ASLR is on by default on all major operating systems. It randomizes memory on a per-module basis. Fine-grained ASLR techniques that additionally randomize memory on the function and/or instruction level were proposed in the literature [19, 29, 40], but have not received widespread adoption yet.

In practice, bypassing standard (i.e., coarse-grained, user-level) ASLR implementations is now common. From an attacker's point of view, disclosing a single pointer that points into a program's shared library is enough to de-randomize the address space [36]. Even fine-grained ASLR implementations cannot withstand sophisticated attacks where the attacker can read code with memory disclosures and assemble a payload on the fly in a JIT-ROP fashion [35].

For kernel-level ASLR, we view kernel memory as another instance of information to hide. From user space, the memory layout of the kernel is not readable and kernel-level ASLR prevents an attacker from knowing the kernel's memory locations. However, previous work showed that it is possible to leak this information via a timing side channel [22].

In general, leaking information by abusing side channels is a viable attack strategy. Typically, an attacker uses a memory corruption to put a program in such a state that she can infer memory contents via timings [10, 17, 33] or other side channels [8]. This way, she can even locate safe areas to which no references exist in unsafe memory.

In the absence of memory disclosures, attackers may still bypass ASLR using Blind ROP (BROP) [8], which can be applied remotely to servers that fork processes several times. In BROP, an attacker sends data that causes a control transfer to another address and then observes how the service reacts. Depending on the data sent the server may crash, hang, or continue to run as normal. By distinguishing all different outcomes, the attacker can infer what code executed and identify ROP gadgets.

In this paper, we specifically focus on safe stacks (which are now integrated in production compilers) and review recent related solutions below.

3.3 Code-Pointer Integrity (CPI)

CPI is a safety property that protects all direct and indirect pointers to code [24]. CPI splits the address space in two. The normal part and a significantly large safe area that stores all code pointers of the program. Access to the safe area from the normal one is only possible through CPI instructions. Additionally, CPI provides every thread with a shadow stack, namely SafeStack, beyond the regular stack. The SafeStack is used for storing return addresses and proven-safe objects, while the regular stack contains all other data. SafeStacks are relatively small but they are all contained in a large safe area, which is hidden at a random location in the virtual address space.

Evans et al. showed how to circumvent CPI and find the safe area by probing using a side channel [17]. Depending on how the safe area is constructed, this attack may require the respawn-after-a-crash property to pull off the attack. This property is only available in (some) servers. Moreover, it is fragile, as it is very easy for an administrator to raise an alarm if the server crashes often. In Section 4, we will introduce an attack that demonstrates how we can efficiently locate CPI's SafeStack in the context of web browsers.

3.4 ASLR-Guard

ASLR-Guard [26] is a recent defense that aims at preventing code-reuse attacks by protecting code addresses from disclosure attacks. It does so by introducing a secure storage scheme for code pointers and by decoupling the code and data regions of executable modules. A core feature is its shadow stack that it uses to separate completely the code pointers from the rest of the data. To efficiently implement this idea, again two separate stacks are used. First, the so called AG-stack which holds only code addresses is used by function calls and returns, exception handlers, etc. The second stack is used for any data operation and ensures that all code pointers and pointers to the AG-stack are encrypted. As a result, an adversary has no way of leaking the location of code images. We discuss the security of this design in Section 4.4.

3.5 Discussion

Information hiding has grown into an important building block for a myriad of defenses. While several attacks on the randomization at the heart of IH are described in the literature, it is still believed to be a formidable obstacle, witness the growing list of defenses that rely on it. Also, since the attacks to date only managed to find secret information occupying a large number of pages, it seems reasonable to conclude, as the authors of ASLR-Guard [26] do, that smaller safe areas are not so vulnerable to probing attacks. In this paper, we show that this is not always true.

4 Breaking Modern Information Hiding

In this section, we introduce two approaches towards uncovering the hidden information. First, we show however careful developers of IH approaches are, pointers to the safe area may still be unexpectedly present in the unsafe area. While this may not represent fundamental problems, there are other issues. Specifically, we show that an attacker may significantly reduce the large randomization entropy for secret data like shadow stacks by making the program spawn many threads in a controlled way, or corrupting the size of the stacks that the program spawns.

4.1 Neglected Pointers to Safe Areas

Safe stack implementations are an interesting target for an attacker and the ability to locate them in a large virtual address space would yield a powerful attack primitive. As an example, consider CPI's SafeStack implementation that is now available in the LLVM compiler toolchain. Recall that the safe stack implementation of CPI moves any potential unsafe variables away from the native stack to make it difficult to corrupt or to gather the exact address of that stack. Any references to the safe stack in global memory that the attacker could leak would therefore break the isolation of SafeStack. Ideally for an attacker, such pointers would be available in program-specific data structures, but we exclude this possibility here and assume that no obvious information disclosure attacks are viable. However, even though the authors diligently try to remove all such pointers, the question is whether there are any references left (e.g., in unexpected places).

For this reason, we analyzed the implementation and searched for data structures that seemed plausible candidates for holding information about the location of stacks. In addition, we constructed a way for an attacker to locate said stacks without relying on guessing. In particular, we examined in detail the *Thread Control Block (TCB)* and *Thread Local Storage (TLS)*.

Whenever the system spawns a new thread for a program, it also initializes a corresponding Thread Control Block (TCB), which holds information about the thread (e.g., the address of its stack). However, once an attacker knows the location of the TCB, she also (already) has the stack location as the TCB is placed on the newly allocated stack of the thread. An exception is the creation of the main thread where the TCB is allocated in a memory region that has been mapped, with `mmap()`, during program initialization. Since the initialization of the program startup is deterministic, the TCB of the main stack is located at a fixed offset from the base address of `mmap()` (which can be easily inferred by leaked pointers into libraries).

Moreover, obtaining the address of the TCB is often easy, as a reference to it is stored in memory and passed to functions of the `pthread` library. While not visible to the programmer, the metadata required to manage threads in multi-threaded applications can also leak the address of the thread stacks. If an attacker is able to obtain this management data, she is also able to infer the location of stacks. Note that the management data is stored in the TCB because threads allocate their own stacks, so they need to free them as well. Furthermore, we found that the TCB also contains a pointer to a linked list with all TCBs for a process, so all stacks can be leaked this way.

Additionally, TLS consists of a static portion and a dynamic portion and the system happens to allocate the static portion of the TLS directly next to the TCB. The TLS portions are managed through the Dynamic Thread Vector (DTV) structure which is allocated on the heap at thread initialization and pointed to by the TCB. Leaking the location of DTV will also reveal the stack location.

Another way to obtain the location of the stacks is using global variables in `libpthread.so`. The locations of active stacks are saved in a double linked list called *stacks_used* which can be accessed if the location of the data section of `libpthread` is known to an attacker.

In summary, our analysis of the implementation reveals that references to sensitive information (in our case safe stacks) do occur in unexpected places in practice. While these issues may not be fundamental, given the complexity of the environment and the operating system, delivering a sound implementation of IH-based defenses is challenging. All references *should* be accounted for in a production defense that regards stack locations as sensitive information. We even argue that any IH-hardening solution (like the one presented in this paper) *should* take implementation flaws of defense solutions such as CPI into account, since they are common and often not under direct control of the solution (e.g., because of external code and libraries).

4.2 Attacks with Thread Spraying

While prior research has already demonstrated that information hiding mechanisms which utilize a large safe area are vulnerable to brute-force attacks [17], our research question is: are *small* safe areas without references to them really more secure than large safe areas? More generally speaking, we explore the limitations of hiding information in an address space and discuss potential attacks and challenges.

In the following, we investigate in detail CPI's SafeStack as an illustrative example. While the safe area itself is very large (dependent on the implementation it may have sizes of 2^{42} or $2^{30.4}$ [17, 25]), a safe stack is only a few MB in size and hence it is challenging to locate it in the huge address space. We analyze the SafeStack implementation available in the LLVM compiler toolchain. As discussed above, the safe stack keeps only safe variables and return addresses, while unsafe variables are moved to an unsafe stack. Hence, an attacker—who has the possibility to read and write arbitrary memory—still cannot leak contents of the safe stack and cannot overwrite return addresses: she needs to locate the safe stack first.

We study if such an attack is feasible against web browsers, given the fact that they represent one of the most prominent attack targets. We thus compiled and linked Mozilla Firefox (version 38.0.5) for Linux using the `-fsanitize=safe-stack` flag of the clang compiler and verified that SafeStack is enabled during runtime. We observed that safe stacks are normally relatively small: each thread gets its own safe stack, which is between 2MB (2^{21} bytes; 2^9 pages) and 8MB (2^{23} bytes; 2^{11} pages) in size. With 28 bits of entropy in the 64-bit Linux ASLR implementation, there are 2^{28} possible page-aligned start addresses for a stack. Hence, an adversary needs at least 2^{19} probes to locate a 2MB stack when sweeping through memory in a brute-force manner. In practice, such an attack seems to be infeasible. For server applications, a brute-force attack would be detectable by external means as it leads to many observable crashes [25].

However, an attacker might succeed with the following strategy to reduce the randomization entropy: while it is hard to find a *single* instance of a safe stack inside a large address space, the task is much easier if she can force the program to generate a lot of safe stacks with a certain structure and then locate just *one* of them. Thus, from a high-level perspective our attack forces the program to generate a large number of safe stacks, a technique we call *thread spraying*. Once the address space is populated with many stacks, we make sure that each stack has a certain structure that helps us to locate an individual stack within the address space. For this, we make use of a technique that we term *stack spraying*, to spray each stack in such a way that we can later easily recognize it. Finally, via a brute-force search, we can then scan the address space and locate a safe stack in a few seconds. In the following, we describe each step in more detail.

4.2.1 Thread Spraying

Our basic insight is that an adversary can abuse legitimate functions to create new stacks, and thereby decrease the entropy. Below, we explain how we performed the thread spraying step in our attack on Firefox. Furthermore, we show that the thread spraying technique is also possible in other applications, namely Chrome and MySQL.

Thread Spraying in Firefox: Our thread spraying in Firefox is based on the observation that an attacker within JavaScript can start *web workers* and each web worker is represented as a stand-alone thread. Thus, the more web workers we start, the more safe stacks are created and the more the randomization entropy drops. Thread spraying may spawn a huge number of threads. In empirical tests on Firefox we were able to spawn up to 30,000 web workers, which leads to 30,000 stacks with a size of 2MB each that populate the address space. In our attack, we implemented this with a malicious website that consists of 1,500 iframes. Each iframe, loading

a webpage from distinct domain name, allows the creation of 20 web workers. As we will show later, forcing the creation of 2,000 or even only 200 stacks is enough in practical settings to locate one of the safe stacks reliably. Fortunately, launching this lower number of concurrent threads is much less resource intensive and the performance impact is small.

Thread Spraying in Chrome: We also tested if Google Chrome (version 45.0.2454.93) is prone to thread spraying and found that Chrome only allows around 60 worker threads in the standard configuration. An investigation revealed that this number is constrained by the total amount of memory that can be allocated for worker threads. When we request more worker threads, the Chrome process aborts as it is unable to allocate memory for the newly requested thread. However, if the attacker has a write primitive, she can perform a data corruption attack [12] and modify a variable that has an effect on the size of the memory space being allocated for worker threads. In Chrome, we found that when we decrease the value of the *global* data variable `g_lazy_virtual_memory`, Chrome will allocate less memory space for a worker thread. The less space allocated, the more worker threads it can spawn. As a result, we were able to spawn up to 250 worker threads, with a default stack size of 8MB, after locating and modifying this data variable, during runtime, in the `bss` section of the Chrome binary.

Thread Spraying in MySQL: We also evaluated the thread spraying attack on the popular MySQL database server (version 5.1.65). Interestingly, MySQL creates a new thread for each new client connection. By default, the maximum number of simultaneous connections is 151 and each thread is created with a stacksize of 256KB. With 151 threads, this amounts to 37.8MB of safe stack area in the memory space which corresponds to spawning just ~19 Firefox or ~5 Chrome worker threads. This would make it hard to perform a successful thread spraying attack. However, as in the Chrome use case above, an attacker with a write primitive can corrupt exactly those variables that constrain the number of threads—using a data-oriented attack [12]. We found that the number of threads in MySQL is constrained by the global variables `max_connections` and `alarm_queue`. Increasing them, allows an attacker to create more connections and thus more threads. Since MySQL has a default timeout of 10 seconds for connections, it may be hard to keep a high number of threads alive simultaneously, but it is just as easy to overwrite the global variables `connect_timeout` and `connection_attrib`, which contains the stack size used when creating a thread for a new client connection. In a simple test we were able to create more than 1000 threads with a stacksize of 8MB.

Protecting the Thread Limits: In some applications, such as Chrome and MySQL, there are global variables that are associated explicitly or implicitly with thread creation. For example, in Chrome there is `g_lazy_virtual_memory` which, if reduced, allows for the creation of more worker threads. Placing these variables in read-only memory can potentially mitigate the thread-spraying attacks, however, it is unclear if the application's behavior is also affected. In Section 5 we present a defense system that protects applications from all attacks discussed in this section without relying on protecting limits associated with thread creation.

4.2.2 Stack Spraying

At this point, we forced the creation of many stacks and thus the address space contains many copies of safe stacks. Next, we prepare each stack such that it contains a signature that helps us to recognize a stack later. This is necessary since we scan in the next step the memory space and look for these signatures in order to confirm that we have indeed found a safe stack (with high probability). In analogy with our first phase, we term this technique *stack spraying*.

From a technical point of view, we realize stack spraying in our attack as follows. Recall that a safe stack assumes that certain variables are safe and this is the case for basic data types such as integers or double-precision floating point values. Moreover, Firefox stores double-precision values in their hexadecimal form in memory. For instance, the number $2.261634509803921 * 10^6$ is stored as `0x4141414141414140` in memory. Additionally, calling a JavaScript function with a double-precision float value as parameter leads to the placement of this value on the safe stack since the system considers it safe. We exploit this feature to (i) fill the stack with values we can recognize and (ii) occupy as much stack space as possible. We therefore use a recursive JavaScript function in every worker which takes a large number of parameters. We call this function recursively until the JavaScript interpreter throws a JavaScript Error (*too much recursion*). As we can catch the error within JavaScript, we create as many stack frames as possible and keep the occupied stack space alive. Of course, other implementations of stack spraying are also possible.

A thread's initial stack space contains various safe variables and return addresses before we call the recursive function the first time. Thus, this space is not controllable, but its size does not vary significantly across different program runs. For example, in our tests the initially occupied stack space had a size of approximately three memory pages ($0x3000$ bytes) in each worker. A sprayed stack frame consists of the values that the recursive function retrieves as parameters and is additionally

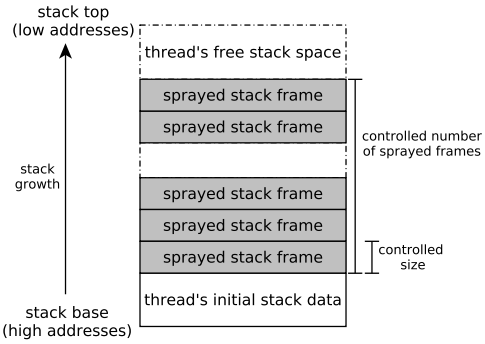

Figure 1: Memory layout of Firefox with CPI's SafeStack filled with sprayed stack frames

interspersed with data intended to reside in a stack frame. As the size of this data is predictable, we can control the size of the stack frame with the number of parameters passed to the recursive function. While the number of sprayed stack frames is controllable via the number of recursive calls, we perform as many recursive calls as the JavaScript interpreter allows.

Figure 1 illustrates the memory layout of a sprayed safe stack after the first two phases of our attack. Since the system keeps safe variables such as double-precision floating point values on the safe stack, the memory can be filled with controlled stack frames which contain user-controlled signatures in a controllable number. Thus, we generate a repetitive pattern on the safe stacks of web workers, which leads to the following observations:

- The probability of hitting a stack during memory probing attempts increases, as the allocated space of a safe stack is filled with a recognizable signature.
- Where safe stacks are not consecutive to each other, they are separated only by small memory regions. Thus, probing with stack-sized steps is possible which reduces the number of probes even further.
- On a signature hit, we can safely read in sprayed frame sized steps towards higher addresses until we do not hit a signature anymore. This tells us that we have reached the beginning of the thread's stack, which we can disclose further to manipulate a return address.

4.3 Scanning methodologies

During our experiments we developed multiple scanning methods fitted to different defense scenarios. In the following we shortly describe the observations leading to the development of each and evaluate them against the targeted defense measures. The first two techniques are targeted at the standard ASLR while the last technique is also successful against an improved version. For our evaluation we assumed that an attacker can not always rely on the stacks being located close to each other. As such we implemented a simple module that can be loaded via `LD_PRELOAD` and forces each call to `mmap`, associated with a stack creation (i.e. `MAP_STACK` provided in the flags argument), to allocate memory at a random address. This means every page is a potential stack base and our first two methods are no longer effective.

4.3.1 Naïve attack on adjacent stacks

The simplest attack is based on the observation that all stacks are allocated close to each other, starting from a randomly chosen base address. To investigate this observation, we spawned 200 worker threads and performed stack spray in each one. We chose a number of parameters for the recursive function such that each sprayed stack frame had a size of one page (4096 bytes). As each thread gets a stack of 2MB in size and individual stacks are grouped closely in memory, we can treat the compound of stacks as a coherent region of approximately 2^{28}B in size. To locate a safe stack we scan towards lower addresses with 2^{28}B sized steps starting at `0x7fffffff000`, the highest possible stack base. As soon as we hit mapped memory we start searching in page sized steps for our sprayed signature. We performed this scan on three Firefox runs and needed only 16755.0 probing attempts on average (consisting of 1241.3 2^{28}B sized probes and 15513.7 page sized probes) to locate a signature and thus a safe stack. While this method is simple to implement and locates stacks with a high probability it has a chance to miss a stack region, if our single probe of a potential stack region hits the gap between two stacks by chance. While retrying with a different starting offset is possible, the next method is more fit for this purpose.

4.3.2 Optimized attack on adjacent stacks

As a variation of our previous method we modified the scanning strategy based on our observations. During three runs with Firefox, we first launched 2,000 worker threads and again performed stack spraying in each of them. Afterwards we conducted our *memory scanning*. The results are shown in Table 1. As our *memory range*, we chose `0x7FxxxxYYYYY0`, whereby the least three significant bytes are fixed (`YYYYY0`) while the fourth and fifth byte remain variable (`xxxx`). This yields a memory range of $2^{16} = 65,536$ addresses: due to 28-bit entropy for top down stack allocations, each of the chosen ranges constitutes a *potential range* for stack allocations. The probability that one of the chosen ranges is *not* a potential stack range is negligibly small.

Table 1: Memory scans in Firefox on eight different ranges after thread and stack spraying was applied- In each range, byte four and five are variable (denoted by ****). Thus, each range consists of $2^{16} = 65536$ addresses. *Mapped* denotes the number of readable addresses, *S-hits* the number of addresses belonging to the safe stack that contain our signature, and *Non S-Hits* represent safe stack addresses not containing our signature. *False S-hits* means that our signature was found at an address not being part of a safe stack.

	Run 1				Run 2				Run 3			
Memory Range	Mapped	S-Hits	Non S-Hits	False S-Hits	Mapped	S-Hits	Non S-Hits	False S-Hits	Mapped	S-Hits	Non S-Hits	False S-Hits
0x7f****000000	878	184	95	0	886	122	138	0	480	73	134	0
0x7f****202020	886	198	74	0	889	154	125	0	482	104	127	0
0x7f****404040	884	182	98	0	890	122	139	1	482	71	129	0
0x7f****606060	890	197	66	0	887	152	123	0	485	107	136	0
0x7f****808080	889	182	92	0	891	123	136	0	482	70	135	0
0x7f****a0a0a0	889	193	60	0	891	152	126	2	482	105	140	0
0x7f****c0c0c0	888	190	86	2	893	122	139	0	485	73	138	0
0x7f****e0e0e0	892	195	64	2	889	151	123	1	485	101	142	1

On average, 753.1 addresses out of 65,536 were mapped for each scan range. The ranges we tested were all potential ranges for safe stacks. 138.5 times a signature was hit, meaning we hit an address being part of a safe stack (*S-hits*). 0.4 times a signature was hit which did not belong to a safe stack. That means we hit a false positive (*false S-hits*). These false hits occur due to the signature being written by the program to non-stack regions which reside in potential stack ranges.

Choosing the three least significant bytes as a fixed value has the advantage of greatly reducing the search space: instead of probing in 2MB steps—which would be necessary to sweep the potential stack locations without a chance of missing a stack—we exploit the observation that the distance between allocated pages varies. Taking this into account leads to the conclusion that stacks are distributed in a way that gives any value of these ranges a roughly equal chance of being part of one stack. While it is not guaranteed to get a hit with this scanning approach, we cover a bigger area in less time. Additionally, in case the first run did not turn up any stack, we are free to retry the scan using a different range. With an increasing number of retries we get close to a full coverage of the scan region, but at the same time we spend less time probing a region without any stacks.

4.3.3 Locating non-adjacent stacks

With our modifications to stack ASLR the methods presented so far have a high chance of missing a stack, because they probe a memory region several times larger than a single stack. Therefore we need to assume no relation between stack locations and are forced to scan for memory of the size of a single stack. With the randomization applied we split the memory into $C = 2^{47}/2^{21} = 2^{26}$ chunks, each representing a possible 2MB stack location. We ignore the fact that some locations are already occupied by modules and heaps, as we are able to distinguish this data from stack data. Also building a complete memory map and then skipping these regions, if possible at all, would take more time than checking for a false stack hit. Without thread spraying we would be forced to locate a single stack, which would mean we would on average need 2^{25} probes. Even with a high scanning speed this would not be feasible. However by spawning more threads we can reduce the number of probes in practice significantly.

We tested two strategies for locating these stacks. In theory every location in the address space has an equal chance of containing a stack, so scanning with a linear sweep with a step size of one stack seems like a valid approach that allows for locating all stacks eventually. However we noticed that the amount of probes required to locate any stack significantly differed from the expected amount. This can be explained by big modules being loaded at addresses our sweep reaches before any stacks. Due to this mechanic we risk sampling the same module multiple times instead of moving on to a possible stack location. As such we employed a different strategy based on a purely random selection of addresses to probe. In total we performed nine measurements and were able to locate a stack with 33,462 probes on average.

4.3.4 Crash-Resistant Memory Scanning

To verify that an attacker can indeed locate CPI's SafeStack when conducting a memory corruption exploit against Firefox, we combined thread and stack spraying with a crash-resistant primitive introduced by recent research [18]. Crash-resistant primitives rely on probing memory while surviving or not causing at all application crashes. Using a crash-resistant primitive, it is possible to probe entire memory regions for mapped pages from within JavaScript and either receive the bytes located at the specified address or a value indicating that the access failed. In case an access violation happens, then the event is handled by an exception handler provided by the application, which eventually survives the crash. An equivalent approach is probing memory using system calls that return an error without crashing when touching unmapped memory. Equipped with crash-resistant primitives, we are free to use any strategy to locate the safe stack without the risk of causing an application crash.

We choose to scan potential ranges which may include safe stacks and hence we choose one of the ranges shown in Table 1. To counteract false positives, we employ a heuristic based on the observation that thread and stack spraying yield stacks of a predetermined size, each of which contains a large number of addresses with our signature. Determining the stack size is easily done after any address inside a stack candidate is found, because we are free to probe higher and lower addresses to locate the border between the stack's memory and neighboring unmapped memory. Once these boundaries are known, it is possible to use memory disclosures to apply the second part of our heuristic. This heuristic is implemented in asm.js as an optimization. As our code is nearly directly compiled to native code it is very fast to execute. Additionally, we mainly need to index an array, with its start address set to the lower boundary, which is a heavily optimized operation in asm.js. If the number of entries in this array matching our sprayed values is above a certain threshold, we conclude that the memory region is indeed a stack. A further optimization we chose was to scan from higher addresses to lower addresses: we observed that stacks are usually the memory regions with the highest address in a process, which means we most likely hit a stack first when scanning from the top.

With the overhead caused by the thread and stack spraying, we are not able to use the full speed offered by the probing primitive [18]. This results in an average probing time of 46s to locate a safe stack (including the time for thread and stack spraying). The speed decrease is mainly due to the fact that we need to keep the threads and stack frames alive. Our attack achieved this by simply entering an endless loop at the end, which leads to degraded performance. However as web workers are designed to handle computational intensive tasks in the background, the browser stays responsive, but the scanning speed is affected.

Tagging safe stacks with user controlled data is not the only option for locating a safe stack. As most free stack space is zeroed out, a simple heuristic can be used to scan for zeros instead of scanning for placed markers. The advantage is that shadow stacks which separate return addresses and data are still locatable. Another possibility is to scan for known return addresses near the base of the stack: as coarse-grained ASLR randomizes shared libraries on a per-module basis and libraries are page aligned, the last twelve bits of return addresses stay the same across program runs and remain recognizable.

Without the overhead caused by the thread and stack spraying, our scanning speed is increased to $16,500$ probes per second. As our approximated scanning method requires $65,536$ scans per run, we are able to finish one sweep in less than 4 seconds. However, this is only the worst case estimation when not hitting any stack. As mentioned before, we are then free to retry using a different value. On average, we are able to locate a safe stack in 2.3 seconds during our empirical evaluation.

4.4 Discussion: Implications for ASLR-Guard

In the following, we discuss the potential of a similar attack against ASLR-Guard [26]. While this defense provides strong security guarantees, it might be vulnerable to a similar attack as the one demonstrated above: as the native stack is used for the AG-stack, we can locate it using our scanning method. If the randomization of AG-stack locations is left to the default ASLR implementation (i.e., the stacks receive the same amount of entropy and potential base addresses), we can use our existing approach and only need to adjust for the size of the stacks (if different) in addition to a different scanning heuristic. This results in a longer scanning time, but if recursion can again be forced by attacker-supplied code, the resulting AG-stack will also increase in size. Combined with the thread spraying attack, we are able to generate a large number of sizable stacks. The major difference is that we are not able to spray a chosen value on the AG-stack. Further research into dynamic code generation might allow for spraying specific byte sequences as return addresses, if code of the appropriate size can be generated. While we can not evaluate the security of ASLR-Guard since we do not have access to an implementation, it seems possible to locate the AG-stack and thus disclose unencrypted code addresses.

Besides the AG-stack, there are two additional memory regions that must be hidden from an attacker. First, the code regions of executable modules are moved to a random location, but they are still potentially readable. As the *mmap-wrapper* is used, they receive an entropy of 28 bits. Since the stacks also receive the same amount of entropy, a similar attack is possible. Scanning can be done in bigger steps if a large executable module is targeted. Second, a safe area called *safe vault* is used for the translation of encoded pointers and it needs to be protected. If either of those structures is located, an adversary is able to launch a code-reuse attack. However, she would be limited to attack types that do not require access to the stack (e.g., COOP [32]). As stated in the paper, an attacker has a chance of 1 in 2^{14} to hit any of these regions with a random memory probe. This results in the possibility of exhausting the search space in roughly one second with the memory probing primitive discussed earlier. Additional steps need to be taken in order to determine the specific region hit, though. This can include signatures for code sections or heuristics to identify the safe vault.

5 Reducing the Odds for Attackers

We developed a mechanism called Authenticating Page Mapper (APM), which hinders attacks probing for the safe areas. Our mechanism is based on a user-level page fault handler authenticating accesses to inactive pages in the safe area and, when possible, also artificially inflating the virtual memory region backing the safe area.

The first strategy seeks to limit the attack surface to active safe area pages in the working set of the application. Since the working set is normally much smaller than the virtual memory size (especially for modern defenses relying on safe areas with sparse virtual memory layouts [13,24]), this approach significantly increases the entropy for the attacker. Also, since a working set is normally stable for real applications [38], the steady-state performance of APM is negligible.

The second strategy ensures an increase in the number of inactive pages not in use by the application, serving as a large trip hazard surface to detect safe area probing attacks with very high probability. In addition, since we randomize the concrete placement of the safe area within the larger inflated virtual memory space, this mitigates the impact of implementation bugs that allow an attacker to disclose the virtual (but not the physical) memory space assigned to the inflated area. Finally, this ensures that, even an attacker attempting to stress-test an application and saturate the working set of a safe area is still exposed to a very large detection surface.

Notice that deterministic isolation, and not hiding, can secure any safe area if properly applied. However, isolation in 64-bit systems has not, yet, been broadly adopted, while CPI's SafeStack is already available in the official LLVM tool-chain [2] and there are discussions for porting it to GCC [3], as well. We therefore seek for a system that rather hardens IH, than fully protects it. To that end, APM stands as a solution until a proper replacement of IH is adopted by current defenses.

5.1 Authenticating Accesses

To authenticate accesses, APM needs to interpose on all the page faults in the safe area. Page faults are normally handled by the OS, but due to the proliferation of virtualization and the need for live migration of virtual machines, new features that enable reliable page fault handling in user space have been incorporated in the Linux kernel [6]. We rely on such features to gain control when an instruction accesses one of the safe areas to authenticate it. To authenticate the access, we rely on unforgeable execution capabilities, such as the faulting instruction pointer and stack pointer, exported by the kernel to our page fault handler and thus trusted in our threat model (arbitrary memory read/write primitives in userland). Our design draws inspiration from recent hardware solutions based on instruction pointer capabilities [37], but generalizes such solutions to generic *execution capabilities* and enforces them in software (in a probabilistic but efficient fashion) to harden IH-based solutions. An alternative option is to use `SIGSEGV` handlers, but this introduces compatibility problems, since applications may have their own `SIGSEGV` handler, faults can happen inside the kernel, etc. On the other hand, `userfaultfd` [6] is a fully integrated technique for reliable page fault handling in user space for Linux.

APM is implemented as a shared library on Linux and can be incorporated in programs protected by CPI, ASLR-Guard, or any other IH-based solution by preloading it at startup (e.g., through the `LD_PRELOAD` environment variable). Upon load, we notify the kernel that we wish to register a user-level page-fault handler for each of the process's safe areas (i.e., using the `userfaultfd` and `ioctl` system calls).

When any of the safe area pages are first accessed through a read or write operation, the kernel invokes the corresponding handler we previously registered. The handler obtains the current instruction pointer (`RIP` on x86-64), the stack pointer, the stack base, and the faulting memory address from the kernel, and uses this information to authenticate the access. Authentication is performed according to defense-specific authentication rules. If the memory access fails authentication, the process is terminated. In the other cases, the handler signals the kernel to successfully map a new zero page into the virtual memory address which caused the page fault.

To support CPI and SafeStack in our current implementation, we interpose on calls to `mmap()` and `pthread_create()`. In particular, we intercept calls to `mmap()` to learn CPI's safe area. This is easily accomplished because the safe area is 4TB and it is the only such mapping that will be made. Furthermore, we intercept `pthread_create()`, which is used to initialize thread-related structures and start a thread, to obtain the address and size of the safe stack allocated for the new thread. In the following subsections, we detail how we implement authentication rules for CPI and SafeStack using our execution capabilities.

5.2 CPI's Authentication Rules

To access a safe area without storing its addresses in data memory, CPI (and other IH-based solutions) store its base address in a CPU register not in use by the application. However, as the number of CPU registers is limited, CPI relies on the segmentation register `gs` available on x86-64 architectures to store the base address. The CPI instrumentation simply accesses the safe area via an offset from that register. Listing 1 shows an example of a safe area-accessing instruction generated by CPI.

```
mov     %gs:0x10(%rax),%rcx
```

Listing 1: x86-64 code generated by CPI to read a value from the safe area.

Since `gs` is not used for any other purpose (it was selected for this reason) and the instrumentation is assumed to be trusted, APM authenticates accesses to the CPI safe area by verifying that the *instruction pointer* points to an instruction using the `gs` register. Therefore, since the attacker needs to access the safe area before actually gaining control of the vulnerable program, only legitimate instructions part of the instrumentation can be successfully authenticated by APM.

5.3 SafeStack's Authentication Rules

Similar to CPI's safe area, SafeStack's primary stack (safe stack) is also accessed through a dedicated register (`RSP` on x86-64 architectures) which originally points to the top of the stack. When new values need to be pushed to it, e.g., due to a function call, the program allocates space by subtracting the number of bytes needed from `RSP`. This occurs explicitly or implicitly through the `call` instruction. Hence, to authenticate safe stack accesses, APM relies on the *stack pointer* (`RSP`) to verify the faulting instruction accesses only the allocated part of the stack. The latter extends from the current value of `RSP` to the base of the safe stack of each thread. We also need to allow accesses the red zone on x86-64.

5.4 Inflating the Safe Area

We inflate safe areas by allocating more virtual address space than it is needed for the area. For example, when a new safe stack is allocated, we can request 10 times the size the application needs. The effect of this inflation is that a larger part of the address space becomes "eligible" for memory-access authentication, amplifying our detection surface. Inflation is lightweight, since the kernel only allocates pages and page-table entries after a page is first accessed.

We implement our inflation strategy for SafeStack (CPI's safe region is naturally "inflated" given the full memory shadowing approach used). To inflate thread stacks, our `pthread_create()` wrapper sets the stack size to a higher value (using `pthread_attr_setstacksize()`). For the main stack, initialized by the kernel, we implement inflation by increasing the stack size (using `setrlimit()`) before the application begins executing. Similar to CPI, we rely on the `mmap()`'s `MAP_NORESERVE` flag to avoid overcommitting a large amount of virtual memory in typical production settings.

To randomize the placement of each safe stack within the inflated virtual memory area, we randomize the initial value of `RSP` (moving it upward into the inflated area) while preserving the base address of the stack and the TCB in place. Since the base address of each stack is saved in memory (as we describe in Section 4), a memory leak can exfiltrate its base address. Our randomization strategy can mitigate such leaks by moving the safe stack working set to a random offset from the base address and exposing guided probing attacks to a large trip hazard surface in between.

6 Evaluation

In this section, we report on experimental results of our APM prototype. We evaluate the our solution in terms of performance, entropy gains (reducing the likelihood attackers will hit the target region), and detection guarantees provided by APM coped with our inflation strategy (authenticating memory accesses to the target region and raising alerts).

We performed our experiments on an HP Z230 machine with an intel i7-4770 CPU 3.40GHz and running Ubuntu 14.04.3 LTS and Linux kernel v4.3. Unless otherwise noted, we configured APM with the default inflation factor of 10x. We repeated all our experiments 5 times and report the median (with little variations across runs).

Performance To evaluate the APM's performance we run the SPEC2006 suite, which includes benchmarks with very different memory access patterns. For each benchmark, we prepared three versions: (1) the original benchmark, denoted as *BL* (Baseline), (2) the benchmark compiled with CPI's SafeStack only, denoted as *SS*, and (3) the benchmark compiled with full CPI support, denoted as *CPI*. Table 2 presents our results. Note that `perlbench` and `povray` fail to run when compiled with CPI, as also reported by other researchers [17]. Therefore, results for these particular cases are excluded from the table.

Not surprisingly, the overhead imposed by APM in all benchmarks and for all configurations (i.e., either compiled using SafeStack or full CPI) is very low. The geometric mean performance overhead increase is only 0.3% for BL+APM, 0.0% for SS+APM and 1.4% CPI+APM.

To confirm our performance results, we evaluated the APM-induced overhead on Chrome (version 45.0.2454.93) and Firefox (version 38.0.5) by running popular browser benchmarks—also used in prior work in the area [21, 23]—i.e., sunspider, octane, kraken, html5, balls and linelayout. Across all the benchmarks, we observed essentially no overhead (0.01% and 0.56% ge-

Apps	BL	BL + APM	SS	SS + APM	CPI	CPI + APM
astar	133.8 sec	1.004x	1.003x	1.002x	0.971x	0.985x
bzip2	82.6 sec	1.003x	1.002x	1.008x	1.039x	1.055x
dealII	229.4 sec	1.008x	1.009x	1.013x	0.887x	0.897x
gcc	19.2 sec	0.978x	0.982x	0.988x	1.368x	1.440x
gobmk	53.6 sec	1.001x	1.020x	1.018x	1.046x	1.046x
h264ref	51.6 sec	1.000x	1.009x	1.013x	1.028x	1.031x
hmmer	113.7 sec	0.996x	1.001x	0.996x	1.063x	1.066x
lbm	248.5 sec	1.002x	1.001x	1.002x	1.154x	1.159x
libquantum	274.1 sec	1.004x	1.015x	1.013x	1.231x	1.227x
mcf	237.4 sec	1.031x	0.998x	0.989x	1.046x	1.045x
milc	349.0 sec	1.009x	0.991x	0.997x	1.012x	1.023x
namd	306.8 sec	1.000x	1.001x	0.997x	1.031x	1.030x
omnetpp	358.8 sec	0.994x	1.017x	1.044x	1.377x	1.472x
perlbench	263.9 sec	1.004x	1.084x	1.091x	——	——
povray	121.3 sec	1.005x	1.092x	1.093x	——	——
sjeng	397.8 sec	1.004x	1.047x	1.051x	1.033x	1.031x
soplex	136.0 sec	1.001x	1.000x	0.951x	1.000x	0.997x
sphinx3	410.6 sec	0.987x	0.997x	0.995x	1.149x	1.138x
xalancbmk	189.0 sec	1.020x	1.042x	1.055x	1.679x	1.782x
geo-mean		1.003x	1.016x	1.016x	1.111x	1.125x

Table 2: SPEC CPU 2006 benchmark results. We present the overhead of hardening state-of-the art defenses with APM. BL and SS refer to *baseline* and *safe stack* (respectively), and CPI refers to CPI's safe area.

ometric mean increas on Chrome and Firefox, respectively). These results confirm that, while APM may introduce a few expensive page faults early in the execution, once the working set of the running programs is fully loaded in memory, the steady-state performance overhead is close to zero. We believe this property makes APM an excellent candidate to immediately replace traditional information hiding on today's production platforms.

Entropy Gains With APM in place, it becomes significantly harder for an adversary to locate a safe area hidden in the virtual address space. To quantify the entropy gains with APM in place, we ran again the SPEC2006 benchmarks in three different configurations, including a parallel shadow stack [14] other than SafeStack and full CPI. We present results for a parallel shadow stack to generalize our results to arbitrary shadow stack implementations in terms of entropy gains. A parallel shadow stack is an ideal candidate for generalization, since its shadow memory-based implementation consumes as much physical memory as a regular stack, thereby providing a worst-case scenario for our entropy gain guarantees.

For each configuration, we evaluated the entropy with and without APM in place and report the resulting gains. The entropy gain is computed as $log_2(VMM/PMM)$, where VMM is the Virtual Mapped Memory size (in pages) and PMM is the Physical Mapped Memory size (in pages). To mimic a worst-case scenario for our entropy gains, we measured PMM at the very end of our benchmarks, when the program has accessed as much memory as possible resulting in the largest resident set (and lowest entropy gains). Table 3 presents our results. Once again, as also reported by other researchers [17], `perlbench` and `povray` are excluded from the CPI configuration.

As expected, our results in Table 3 show that lower stack usage (i.e., lower PMM) results in higher entropy gains. Even more importantly, the entropy gains for CPI-enabled applications are substantial. In detail, we gain 11 bits of entropy even in the worst case (i.e., `xalancbmk`). In other cases, (e.g., `bzip2`) the entropy gains go up to 28 bits of entropy.

We find our experimental results extremely encouraging, given that, without essentially adding overhead to CPI's fastest (but low-entropy) implementation, our techniques can provide better entropy than the slowest (probabilistic) CPI implementation [25]. SafeStack's entropy gains are, as expected, significantly lower than CPI's, but generally (only) slightly higher than a parallel shadow stack. In both cases, the entropy gains greatly vary across programs, ranging between 2 and 11 bits of entropy. This is due to the very different memory access patterns exhibited by different programs. Nevertheless, our strategy is always effective in nontrivially increasing the entropy for a marginal impact, providing a practical and immediate improvement for information hiding-protected applications in production.

Apps	Parallel Shadow Stack				SafeStack				CPI's (safe region)			
	VMM	PMM	EG	DG	VMM	PMM	EG	DG	VMM	PMM	EG	DG
astar	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	201668	> 12 bits	99.98 %
bzip2	2048	4	9 bits	99.98 %	2048	1	11 bits	100.00 %	1 GP	4	28 bits	100.00 %
gcc	2048	112	> 4 bits	99.45 %	2048	8	8 bits	99.96 %	1 GP	121314	> 13 bits	99.99 %
gobmk	2048	27	> 6 bits	99.87 %	2048	7	> 8 bits	99.97 %	1 GP	5813	> 17 bits	100.00 %
h264ref	2048	5	> 8 bits	99.98 %	2048	2	10 bits	99.99 %	1 GP	6994	> 17 bits	100.00 %
hmmer	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	36616	> 14 bits	100.00 %
lbm	2048	2	10 bits	99.99 %	2048	1	11 bits	100.00 %	1 GP	2	> 29 bits	100.00 %
libquantum	2048	1	11 bits	100.00 %	2048	2	10 bits	99.99 %	1 GP	66911	> 13 bits	99.99 %
mcf	2048	2	10 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	5107	> 17 bits	100.00 %
milc	2048	2	10 bits	99.99 %	2048	1	11 bits	100.00 %	1 GP	20017	> 15 bits	100.00 %
namd	2048	10	> 7 bits	99.95 %	2048	2	10 bits	99.99 %	1 GP	109	> 23 bits	100.00 %
omnetpp	2048	34	> 5 bits	99.83 %	2048	10	> 7 bits	99.95 %	1 GP	171316	> 12 bits	99.98 %
perlbench	2048	491	> 2 bits	97.60 %	2048	446	> 2 bits	97.82 %	—	—	—	—
povray	2048	7	> 8 bits	99.97 %	2048	4	9 bits	99.98 %	—	—	—	—
sjeng	2048	132	> 3 bits	99.36 %	2048	26	> 6 bits	99.87 %	1 GP	2	> 29 bits	100.00 %
soplex	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	31673	> 15 bits	100.00 %
sphinx3	2048	12	> 7 bits	99.94 %	2048	2	10 bits	99.99 %	1 GP	11334	> 16 bits	100.00 %
xalancbmk	2048	496	> 2 bits	97.58 %	2048	494	> 2 bits	97.59 %	1 GP	316838	> 11 bits	99.97 %

Table 3: Entropy gains with our defense. VMM, PMM, EG, and DG refer to *Virtual Mapped Memory*, *Physical Mapped Memory*, *Entropy Gains* and *Detection Guarantees* (respectively). VMM and PMM are measured in number of pages. EG is given by $log_2(VMM/PMM)$. DG is given by $(1 - PMM/(VMM * inflation\_factor)) * 100$, where the *inflation_factor* is set to default 10x for stacks and 1x for the already huge CPI's safe region. GP stands for *giga pages*, i.e., $1024 * 1024 * 1024$ regular pages. A regular page has a size of 4096 bytes.

Detection Guarantees Table 3 also illustrates the detection guarantees provided by APM when coped with the default 10x inflation strategy. The detection guarantees reflect the odds of an attacker being flagged probing into the inflated trip hazard area rather than in any of the safe pages mapped in physical memory. As shown in the table, APM offers very strong detection guarantees across all our configurations. Naturally, the detection guarantees are stronger as the size of the inflated trip hazard area (i.e., $VMM * inflation\_factor - PMM$) increases compared to the resident size (i.e., PMM). The benefits are, again, even more evident for CPI's sparse and huge safe area, which registered 100% detection guarantees in almost all cases. Even in the worst case (i.e., *xalancbmk*), CPI retains 316,838 trip hazard pages at the end of the benchmark, resulting in 99.97% detection guarantees.

To lower the odds of being detected, an attacker may attempt to force the program to allocate as many safe area physical pages as possible, naturally reducing the number of trip hazard pages. We consider the impact of this scenario in Firefox, with a JS-enabled attacker spraying the stack to bypass APM. Figure 2 presents our results for different inflation factors assuming an attacker able to spray only the JS-visible part of the stack (1MB) or the entire stack to its limit (2MB). As shown in the figure, in both cases, APM provides good detection guarantees for reasonable values of the inflation factor and up to 95% with a 20x inflation (full spraying setting). Even in our default configuration, with a 10x inflation, APM offers adequate detection guarantees in practice (90% for the full spraying setting).

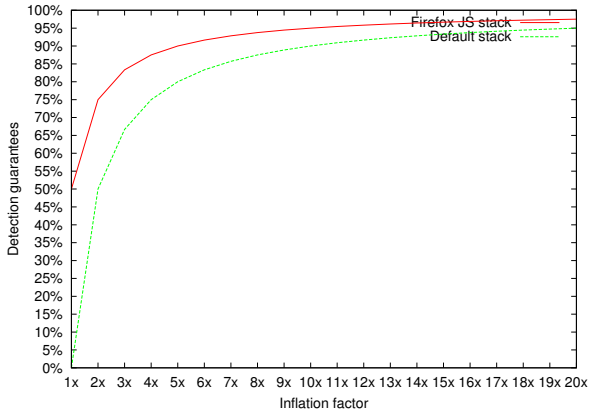

Figure 2: Effect of stack spraying (JS-visible or default stack) on our detection guarantees (DGs) across different inflation factors.

Limitations APM aims at hardening IH, but does not guarantee that a defense based on IH is fully protected against arbitrary attacks. Defenses that rely on IH should properly isolate the safe area to preserve the integrity and/or confidentiality of sensitive data. In the absence of strong (e.g., hardware-based) isolation, however, APM can transparently raise the bar for attackers, since it can offer protection without programs being aware of it (no re-compilation or binary instrumentation is needed). Nevertheless, certain attacks can still reduce the entropy and the detection guarantees provided by APM. For example, an attacker may be able to locate the base address of an inflated safe area by exploiting an implementation flaw or the recent allocation oracle side channel [28].

While the entropy is reduced, the trip hazard pages still deter guided probing attacks in the inflated area. However, if an implementation flaw or other side channels were to allow an attacker to leak a pointer to an active safe area page in use by the application (e.g., `RSP`), APM would no longer be able to detect the corresponding malicious access, since such page has already been authenticated by prior legitimate application accesses.

7 Conclusion

Information hiding is at the heart of some of the most sophisticated defenses against control-flow hijacking attacks. The assumption is that an attacker will not be able to locate a small number of pages tucked away at a random location in a huge address space if there are no references to this pages in memory. In this paper, we challenge this assumption and demonstrate that it is not always true for complex software systems such as Mozilla Firefox. More specifically, we examined CPI's SafeStack since it is considered to be the state-of-the-art defense. In a first step, we analyzed the implementation and found that there were still several pointers to the hidden memory area in memory. An attacker can potentially abuse a single such pointer to bypass the defense. More seriously still, the protection offered by high entropy is undermined by thread spraying—a novel technique whereby the attacker causes the target program to spawn many threads in order to fill the address space with as many safe stacks as possible. Doing so reduces the entropy to the point that brute-force attacks become viable again. We demonstrated the practicality of thread spraying by way of an attack against Firefox, Chrome and MySQL, protected with CPI's SafeStack.

To mitigate such entropy-reducing attacks, we propose an IH hardening strategy, namely APM. Based on a user-space page fault handler, APM allows accessing of pages on demand only and vets each first access to a currently guarded page. The additional protection provided by the page fault handler greatly improves the pseudo-isolation offered by information hiding, making it a concrete candidate to replace traditional information hiding in production until stronger (e.g., hardware-based) isolation techniques find practical applicability. Most notably, our approach can be used to harden existing defenses against control-flow hijacking attacks with barely measurable overhead.

Acknowledgements

We thank the reviewers for their valuable feedback. This work was supported by Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI "Dowsing" project, by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457, and by ONR through grant N00014-16-1-2261. Any opinions, findings, conclusions and recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US Government, or the ONR.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

References

[1] Applications using older atl components may experience conflicts with dep. `https://support.microsoft.com/en-us/kb/948468`.

[2] Clang's SafeStack. `http://clang.llvm.org/docs/SafeStack.html`.

[3] Discussion for porting SafeStack to GCC. `https://gcc.gnu.org/ml/gcc/2016-04/msg00083.html`.

[4] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. Opus: online patches and updates for security. In *USENIX Security '05*.

[5] ANDERSEN, S., AND ABELLA, V. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention, 2004. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

[6] ARCANGELII, A. Userfaultfd: handling userfaults from userland.

[7] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security '14*.

[8] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *IEEE S&P '14*.

[9] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: A new class of code-reuse attack. In *ASIA CCS '11*.

[10] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE S&P '16*.

[11] CARLINI, N., AND WAGNER, D. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security '14*.

[12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security '05*.

[13] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *ASIA CCS '15*.

[14] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *ASIA CCS '15*.

[15] DAVI, L., LIEBCHEN, C., SADEGHI, A. R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS '15*.

[16] DAVI, L., SADEGHI, A.-R., LEHMANN, D., AND MONROSE, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security '14*.

[17] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE S&P '15*.

[18] GAWLIK, R., KOLLENDA, B., KOPPE, P., GARMANY, B., AND HOLZ, T. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS '16*.

[19] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security '12*.

[20] GÖKTAŞ, E., ATHANASOPOULOS, E., POLYCHRONAKIS, M., BOS, H., AND PORTOKALIDIS, G. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security'14*.

[21] HALLER, I., GÖKTAŞ, E., ATHANASOPOULOS, E., PORTOKALIDIS, G., AND BOS, H. Shrinkwrap: Vtable protection without loose ends. In *ACSAC '15*.

[22] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *IEEE S&P '13*.

[23] JANG, D., TATLOCK, Z., AND LERNER, S. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *NDSS '14*.

[24] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer Integrity. In *OSDI '14*.

[25] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., AND SONG, D. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. In *IEEE S&P '15*.

[26] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *CCS '15*.

[27] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque Control-Flow Integrity. In *NDSS '15*.

[28] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *USENIX Sec '16*.

[29] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P '12*.

[30] PAX TEAM. Address Space Layout Randomization (ASLR), 2003. `pax.grsecurity.net/docs/aslr.txt`.

[31] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under unix. In *USENIX ATC '95*.

[32] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE S&P '15*.

[33] SEIBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *CCS '14*.

[34] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07*.

[35] SNOW, K. Z., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., MONROSE, F., AND SADEGHI, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE S&P '13*.

[36] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *EuroSec EWSS '09*.

[37] VILANOVA, L., BEN-YEHUDA, M., NAVARRO, N., ETSION, Y., AND VALERO, M. Codoms: Protecting software with code-centric memory domains. In *ISCA '14*.

[38] VOGT, D., MIRAGLIA, A., PORTOKALIDIS, G., BOS, H., TANENBAUM, A. S., AND GIUFFRIDA, C. Speculative memory checkpointing. In *Middleware '15*.

[39] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP '93*.

[40] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS '12*.

[41] XI CHEN, A. S., DENNIS ANDRIESSE, H. B., AND GIUFFRIDA, C. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS '15*.

[42] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORM, T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND INC, G. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE S&P '09*.

[43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE S&P '13*.

[44] ZHANG, M., AND SEKAR, R. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. In *ACSAC '15*.