Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks

Mauro Conti*, Stephen Crane‡, Lucas Davi†, Michael Franz‡, Per Larsen‡,
Christopher Liebchen†, Marco Negro†, Mohaned Qunaibit‡, Ahmad-Reza Sadeghi†

†CASED, Technische Universität Darmstadt, Germany
‡University of California, Irvine
*University of Padua, Italy

Abstract

Adversaries exploit memory corruption vulnerabilities to hijack a program's control flow and gain arbitrary code execution. One promising mitigation, control-flow integrity (CFI), has been the subject of extensive research in the past decade. One of the core findings is that adversaries can construct Turing-complete code-reuse attacks against coarse-grained CFI policies because they admit control flows that are not part of the original program. This insight led the research community to focus on fine-grained CFI implementations.

In this paper we show how to exploit heap-based vulnerabilities to control the stack contents including security-critical values used to validate control-flow transfers. Our investigation shows that although program analysis and compiler-based mitigations reduce stack-based vulnerabilities, stack-based memory corruption remains an open problem. Using the Chromium web browser we demonstrate real-world attacks against various CFI implementations: 1) against CFI implementations under Windows 32-bit by exploiting unprotected context switches, and 2) against state-of-the-art fine-grained CFI implementations (IFCC and VTV) in the two premier open-source compilers under Unix-like operating systems. Both 32 and 64-bit x86 CFI checks are vulnerable to stack manipulation. Finally, we provide an exploit technique against the latest shadow stack implementation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

stack corruption; control-flow integrity; code-reuse attacks

1. MOTIVATION

Computer systems still run vast amounts of legacy software written in unsafe languages such as C and C++. In the pursuit of efficiency and flexibility, languages in the C family eschew safety features such as automatic memory management, strong typing, and overflow detection. As a result, programming errors can lead to memory corruption and memory disclosure vulnerabilities that are exploited by attackers—often with severe consequences.

Although defenses such as stack canaries, data execution prevention (DEP), and address space layout randomization (ASLR) have raised the bar for attackers, they do not stop sophisticated exploits against complex software.

Since DEP prevents code injection, code reuse has become a key technique in modern exploits that bypass all current mitigations. Code-reuse attacks repurpose legitimate instruction sequences (called gadgets) that are present in program memory to avoid the need for code injection.

Control-flow integrity (CFI) [1, 3], code-pointer integrity (CPI) [32], and code randomization [33] have emerged as the most promising improvements over the exploit mitigations used today. CFI ensures that a program's control flow follows a predefined control-flow graph, CPI protects code pointers from being overwritten, and randomization hides the code layout.

On one hand, all these approaches improve resilience against code-reuse attacks. On the other hand, practical implementations of these techniques must provide an appropriate balance between efficiency, security, and compatibility with legacy software like browsers, document readers, and web servers. The need to meet these constraints often leads to compromises that enable attacks on implementations of CFI [12, 38, 55, 57, 58], fine-grained code randomization [17, 43, 45], and CPI [21].

We focus on CFI implementations because CFI has been the subject of intensive research in the past decade. Moreover, Microsoft and Google recently added CFI support to the most popular C/C++ compilers [36, 50].

Goal and contributions. Our main objective concerns the pitfalls that must be taken into account when implementing fine-grained CFI defenses. We show that failing to do so leads to vulnerabilities that can be exploited to undermine the formal security properties of CFI [2]. In particular, we discovered that CFI implementations in two major compilers—*indirect function call checks* (IFCC) in LLVM, and *virtual table verification* (VTV) in GCC [50]—can be bypassed in a realistic adversary model. In principle, both CFI schemes provide strong protection, and even resist the latest code-reuse attack techniques such as COOP (counterfeit object-oriented programming) [40]. However, the im-

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813671.

plemented CFI checks contain weaknesses due to unanticipated interactions with optimizations applied by the compiler. Since both compilers are the foundation for many open-source projects, our findings affect a wide range of applications.

We also demonstrate a new attack that exploits unprotected context switches (i.e., user-mode return addresses) between user and kernel mode to bypass any CFI implementation for Windows on x86 32-bit that does not protect these context switches. In general, return addresses can be protected through a shadow stack [16], however, this applies only to return addresses that are used by the application and not to those which are used by kernel. By obtaining control over multiple program threads we can create race conditions to undermine control-flow checks.

Our exploits contradict the widely held belief that stack corruption is a solved problem. For instance, both IFCC and VTV focus on heap-based attacks, like virtual-function table manipulation, assuming that the program stack and the return addresses it contains cannot be manipulated due to randomization and stack canaries [50]. However, this assumption is only valid for most stack-related vulnerabilities such as the classic stack-based buffer overflow [4]. We revisit this assumption and demonstrate new attacks against the stack using memory-corruption vulnerabilities that are unrelated to the stack. We call our combined attacks *StackDefiler*, which not only pose a severe threat to CFI implementations but also question the security of stack protections such as StackGuard [13] and StackArmor [11, 13] (see Section 6.2.1).

In summary, our contributions are as follows:

- **Attack on stack-spilled registers**. We found that compiler optimizations spill critical CFI pointers on the stack. To confirm that the weaknesses we identified are exploitable in practice, we created a proof-of-concept implementation to exploit this flaw and bypass Google's fine-grained forward-edge CFI implementation [50]. Both IFCC and VTV suffer from this weakness on x86 (32 and 64-bit) systems. To mitigate our attack, we also developed a patch for IFCC and evaluated its efficiency.

- **Attack on user-mode return addresses on the stack**. We show how to bypass CFI by overwriting the user-mode return address that is used by the kernel to return from a system call. Our attack requires no kernel privileges, and is within the threat model of all CFI schemes, i.e., arbitrary read and write of data memory in user mode. We present a proof-of-concept implementation of our attack that uses multi-threading to bypass CFI implementations on Windows not protecting the user-mode return address. This attack applies to operating systems where user-mode return addresses are pushed on the stack during a system call (Windows 32-bit).

- **Attack on shadow stacks**. Shadow stacks are used in CFI to protect return instructions. However, protecting shadow stacks is expensive [16, 39] unless they are protected by hardware, which is not always available. We show that shadow stacks not protected by hardware are vulnerable to memory disclosure, and provide a proof-of-concept implementation of an exploit against the latest shadow stack implementation [16].

- **Attack the stack from the heap**. We present a new technique to leak the address of the program stack allowing us to reliably alter stack content. While there has been some effort to protect the stack (StackGuard [13] and StackArmor [11]), these defenses are based on the implicit assumption that attacks on the stack use stack-based vulnerabilities. We demonstrate stack attacks that are based on heap vulnerabilities, and hence, undermine this assumption. This has direct impact on defenses that rely on the stack integrity.

2. CODE-REUSE ATTACKS AND CFI

Code-reuse attacks require the target program to contain memory corruption vulnerabilities such as buffer overflows that the adversary can exploit to hijack the control flow and redirect execution to existing code fragments already available in an application's memory space. A prominent attack technique is return-oriented programming (ROP), which is based on maliciously combining and executing a chain of short code sequences of an application [44]. The key element is the return instruction which serves as connecting link for the various code sequences.

One of the most promising defense technique to thwart code-reuse attacks is control-flow integrity [1, 3]. The main idea of CFI is to compute an application's control-flow graph (CFG) prior to execution, and then monitor its run-time behavior to ensure that the control flow follows a legitimate path in the CFG. The destinations of forward edges of the CFG (indirect call and jump instructions) are computed during a static analysis phase and enforced at runtime, e.g., through label checking. The same is possible for backward edges (return instructions), however, static enforcement can lead to an exploitable imprecision [9], because one function can be called by a large number of functions. Therefore, CFI utilizes a *shadow stack* [3, 16] which stores the benign return targets securely on a separate stack and enforces that returns must target the calling function[1]. Any deviation from the CFG leads to the termination of the application. Validating all indirect control-flow transfers can have a substantial performance impact that prevents widespread deployment. For instance, when validating forward and backward edges, the average run time overhead is 21% for the initially proposed CFI [1] and 13.7% for state-of-the-art solutions (4.0% for forward [50] and 9.7% for backward edges [16]).

Several CFI frameworks aim at reducing the run-time overhead by enforcing coarse-grained policies. There is no clear definition in the literature with respect to the terms *fine* and *coarse-grained* CFI policy. However, the general understanding of a *fine-grained* CFI policy is that only branches intended by the programmer are allowed. In contrast, a *coarse-grained* CFI policy is more relaxed and might allow indirect branches to target the start address of any function. For instance ROPecker [12] and kBouncer [38] leverage the branch history table of modern x86 processors to perform a CFI check on a short history of executed branches. Zhang and Sekar [58] and Zhang et al. [57] applied coarse-grained CFI policies using binary rewriting to

[1]In practice this policy is relaxed by allowing the program to return to *any* of the active call-sites for compatibility with exception handling.

protect COTS binaries. Relaxing the CFI policies (or introducing imprecision to the CFG) has the downside of enabling the adversary to launch code-reuse attacks within the enforced CFG. Consequently, coarse-grained variants of CFI have been repeatedly bypassed [10, 18, 25, 26]. This important insight has recently turned the research focus towards fine-grained CFI implementations [1, 16, 50]. Particularly, we consider the implementations of CFI for the LLVM and GCC compiler by Tice et al. [50], called indirect function call check (IFCC) and virtual table verification (VTV). IFCC verifies indirect calls by ensuring that the call destination is within a so-called *jump table* which contains jump instructions to valid targets for the indirect call. The set of valid call-targets is determined at compile time. VTV only protects virtual function calls. At compile time the compiler derives the set of valid vTables and inserts checks at every virtual call site. These checks verify that the virtual table used for the function call is valid for the current object.

3. THREAT MODEL AND ASSUMPTIONS

Our threat model captures the capabilities of real-world attacks, and is in line with the common threat model of CFI [3], as well as with the prior offensive work [20, 40, 43, 45].

Adversarial Capabilities.

- **Memory read-write:** The target program contains a memory-corruption vulnerability that allows the adversary to launch a run-time exploit. In particular, we focus on vulnerabilities that allow the adversary to read (information disclosure) and write arbitrary memory. Such vulnerabilities are highly likely as new vulnerabilities are being constantly reported. Common examples are use-after-free errors [49].
- **Adversarial Computation:** The adversary can perform computations at run time. Many modern targets such as browsers, Flash, Silverlight, and document viewers, as well as server-side applications and kernels allow the adversary to perform run-time computations. Real-world attacks on client-side applications typically utilize a scripting environment to instantiate and perform a run-time exploit. Additionally, the adversary can use the scripting engine to generate multiple execution threads.

Defensive Assumptions.

- **Non-Executable Memory:** The target system enforces data execution prevention (DEP) [35]. Otherwise the adversary could directly manipulate code (e.g., overwriting CFI checks), or inject new malicious code into the data section of a program. The adversary is therefore limited to code-reuse attacks.
- **Randomization:** The target system applies address space layout randomization (ASLR).
- **Shadow Stack:** We do not have access to the implementation of shadow stacks [1, 16]. Therefore, we assume the presence of an adequate shadow stack implementation.

4. STACKDEFILER

Our attacks are based on modifying data on the stack. Hence, as a first step, in the presence of ASLR, we must disclose the address of the stack. We stress that we do not rely on *stack-based* vulnerabilities to attack the stack. Instead, we used *heap-based* vulnerabilities in our exploits. We observe that an adversary with the ability to disclose arbitrary memory can get a stack address by recursively disclosing data pointers (see Section 5.2). Attacking values on the stack is challenging, because (i) only certain functions will write critical data to the stack, and (ii) the lifetime of values on the stack is comparatively short, i.e., generally during the execution of a function. Nevertheless, we are able to manipulate targeted values on the stack.

In the following we give a high-level description of our attacks. For this we discuss three different stack-corruption techniques that allow us to bypass the CFI implementations we examined.

4.1 Corrupting Callee-Saved Registers

To maximize the efficiency of a program, the compiler tries to maximize the use of CPU registers, instead of using the (slower) main memory. The compiler performs register allocation to keep track which registers are currently in use and to which it can assign new values. If all registers are in use, but a register is required to perform a computation, the compiler temporarily saves the content of the register to the stack. When a function (the caller) calls another function (the callee), the callee cannot determine which of the caller's registers are used at the moment of the call. Therefore, the callee saves all registers it needs to use during its execution temporarily on the stack. These saved registers are called *callee-saved registers.* Before the callee returns to the caller it restores all callee-saved registers. While the registers are saved, the adversary can change the values on the stack and therefore corrupt the callee-saved registers. This becomes a severe problem if the caller uses the restored (and potentially corrupted) registers for CFI checks and can affect all architectures where the application binary interface (ABI) specifies the concept of callee-saved registers.

We found that two CFI implementations, IFCC and VTV [50], are vulnerable to this kind of attack. As we will argue in the following, this threat becomes even more crucial for applications that are compiled with position-independent code (PIC) for architectures that do not support program-counter relative (PC-relative) addressing, such as x86 32-bit.

Position-Independent Code

On Unix-like systems, including Mac OS X and Linux, ASLR compatible binaries contain position-independent code (PIC). Position independence means that all code references are relative to the program counter (PC). This allows the dynamic loader to load the binary at an arbitrary base address without relocating it.

However, Intel x86 processors running 32-bit code do not directly support PC-relative addressing. As a workaround, PIC on x86 requires the program to obtain the current value (i.e., the absolute address) of the program counter dynamically at run time. Once this address is known, the program can perform PC-relative references. At assembly level this is implemented by executing a call to the subsequent instruction. The call automatically loads the return address onto the stack, where the return address is simply the absolute

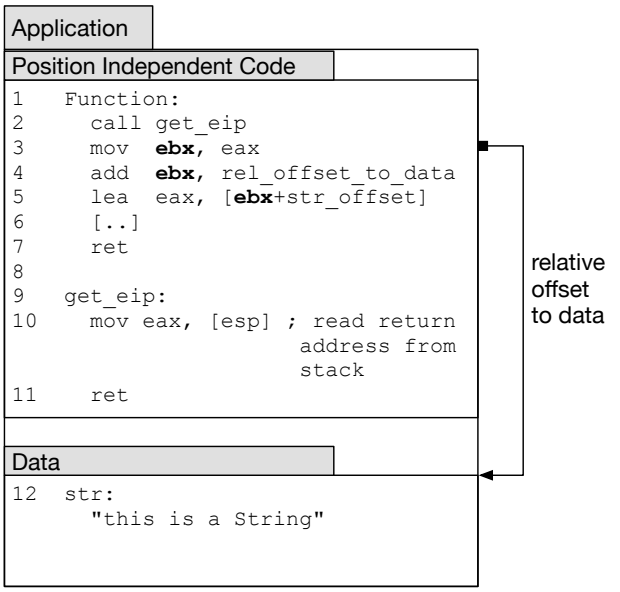

Figure 1: Application compiled with position-independent code. To get the absolute address of `str` the compiler emits instructions that first receive the absolute address of `Function` at run time. The absolute address of `str` is then calculated by adding the relative offset between `Function` and `str`, calculated by the compiler, to the absolute address of `Function`.

address of the subsequent instruction. Hence, the program can obtain its current program counter by simply popping the return address off the stack in the subsequent instruction. Once the program counter is loaded into a register, an offset is added to form the position-independent reference.

Figure 1 illustrates how position-independent code references the global string variable `str` in the data section (Line 12). At function entry, the function calls `get_eip()` (Line 2). This function (Line 9) only reads the return address from the stack (Line 10), which is the address of the instruction following the call of `get_eip()` (Line 3). Next, the result is moved into the `ebx` register (Line 3). We noticed that both LLVM and GCC primarily use the `ebx` register to compute position-independent references (Line 5).

Subsequently, the program can perform PC-relative addressing to access the global string variable: the `add` instruction adds the relative offset between the data section and the current function to `ebx` which now holds a pointer to the data section (Line 4). Finally, the offset of the string within the data section is added to `ebx` and the result (address of the string variable) is saved in the `eax` register (Line 5).

On x86 32-bit platforms PIC becomes a vulnerability for CFI, because the global CFI policies are addressed through the `ebx` register. Since `ebx` is a callee-saved register it is spilled on the stack by all functions that perform CFI checks.

4.2 Corrupting System Call Return Address

Fine-grained CFI as proposed by Abadi et al. [1] validates the target address of every indirect branch. Valid forward edges of the CFG are determined using static analysis and are enforced through label checking. A shadow stack is used to verify the backward edges of the CFG. We noticed that user-mode CFI only instruments user-mode applications and not the kernel. In general, this makes sense because the kernel isolates itself from user-mode applications, and hence, is considered trusted. However, we discovered a way to bypass CFI without compromising the kernel. In particular, we exploit the fact that the kernel reads the return address used to return from a system call to the user mode from the user-mode stack.

On x86 32-bit a special instruction—`sysenter`—was introduced to speed up the transition between user and kernel mode [30]. The `sysenter` instruction does not save any state information. Therefore Windows saves the return address to the user-mode stack before executing `sysenter`. After executing the system call, the kernel uses the saved return address to switch back to user mode. This opens a small window of time between the return address being pushed on the stack and the kernel reading it to switch back to user mode. We use a second, concurrent thread that exploits this window to overwrite the saved return address. Hence, when returning from a system call the kernel uses the overwritten address. This allows the adversary to set the instruction pointer to an arbitrary address and bypass CFI policy checks.

CFI在user-mode下检查，在kernel下不检查

Note that this attack works within the adversary model of CFI because we never modify existing code, nor corrupt the kernel, or tamper with the shadow stack, but we exploit a missing check of a code pointer that can be controlled by the adversary.

The 64-bit x86 architecture uses a different instruction, called `syscall`, to switch from user to kernel mode. This instruction saves the user-mode return address into a register, thus preventing the adversary from changing it. However, even 64-bit operating systems provide an interface for `sysenter` to be compatible with 32-bit applications. Hence, 32-bit applications that are executed in 64-bit operating systems remain vulnerable. Another pitfall of 64-bit x86 is that it partially deprecates memory segmentation, hence, the shadow stack can no longer be completely protected via hardware.

Hence, the shadow stack can no longer be completely protected via hardware. As a consequence the protection of the shadow stack relies on information hiding or less efficient software-fault isolation techniques.

4.3 Disclosing the Shadow Stack Address

Dang et al. [16] survey the different implementations of shadow stacks and their performance costs. One observation is that a *parallel shadow stack*, i.e., a shadow stack located at a constant offset to the normal stack, provides the best performance. However, as we demonstrate in Section 5.2 the adversary can leak the address of the normal stack and therefore compute the address of the shadow stack.

Another shadow stack technique utilizes the *thread-local storage* (TLS), a per-thread memory buffer usually used to store thread-specific variables. In the following we discuss potential implementation pitfalls of this approach. However, we have not implemented this attack due to the unavailability of implementations in public domain. TLS is addressed through a segment register. Although segmentation is no longer available under x86 64-bit, segment registers are still present and can be used to address memory. In general,

a TLS-based shadow stack implementation first loads the shadow-stack pointer into a general purpose register. Next, this register is used to save the return address on the shadow stack [1, 16]. However, we did not find any evidence that the registers used during this operation are cleared afterwards. Hence, the address of the shadow stack may be leaked when a function pushes the used register on the stack. Further, an application might hold a reference in one of its memory objects that can be leaked to disclose the memory address of TLS and the shadow stack.

5. STACKDEFILER IMPLEMENTATION

We now turn our attention to the practical implementation of the previously described attacks. To prove the effectiveness of these attacks we start from real-world vulnerabilities. For our proof-of-concept implementation of the attacks we chose the Chromium web browser because it is available for all common operating systems, and implements state-of-the-art heap and stack software defenses. We stress that our attacks also apply to other applications that provide the adversarial capabilities we outlined in Section 3. This includes document viewers, Flash, Silverlight, server-side applications and kernels. We re-introduced an older software vulnerability (CVE-2014-3176) in the most recent version of Chromium (v44.0.2396.0)—we did not make any further changes to the source code.

To prove that stack spilled registers pose a severe threat to modern, fine-grained forward-edge CFI implementation we compiled Chromium with IFCC for 32 and 64-bit on Ubuntu 14.04 LTS. We disassembled IFCC and VTV protected applications to verify that they are vulnerable to stack-spilling attacks on other operating systems (Unix and Mac OS X) as well. We implemented our attack against the initial proposed CFI [1] on a fully patched Windows 7 32-bit system. Since the implementation of the originally proposed CFI [1] is not available, we assume that fine-grained CFI with a secure shadow stack and construct our attack under the constrains given by the paper.

After giving a short introduction to browser exploitation, we give a detailed description of our proof-of-concept exploits that bypass existing CFI implementations.

5.1 Attacking a Web Browser

While adversary-controlled JavaScript in browsers is generally sandboxed by enforcing type and memory safety, the runtime used to interface the browser and web contents is not. Performance critical parts of the JavaScript runtime library are written in lower level, unsafe languages, e.g., C++. The usage of C++ opens the door for memory-related security vulnerabilities. Memory corruption is then used to manipulate the native representation of website objects, which cannot be done directly from JavaScript code. Next, we explain how this can be exploited to read arbitrary memory and hijack the program control flow.

5.1.1 *Information Disclosure*

Websites create a variety of objects using the browser's scripting engine. These objects are stored consecutively in memory. For instance, the native representation of an array object is usually a C++ object with two fields: the length of the array followed by its starting address, as shown in Figure 2. A JavaScript program can read the contents of the array by using the runtime interface provided by the native C++ object. To ensure memory safety, the native read function uses the saved array length to ensure that the JavaScript program does not access memory outside the arrays bounds. By using a memory corruption vulnerability, the adversary can overwrite the array length in the native representation of the array object with a larger value, as shown in Step ❶. This allows the adversary to read the memory beyond the original array boundaries using normal JavaScript code (Step ❷) and disclose the contents of a subsequent C++ object.

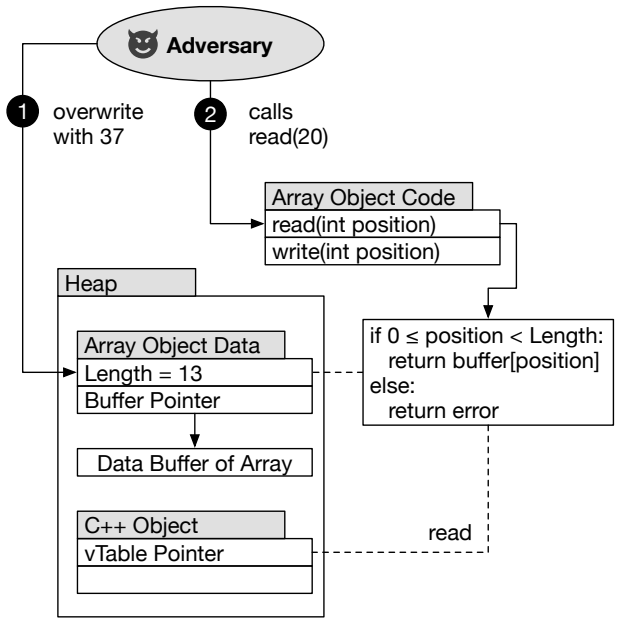

Figure 2: The adversary can overwrite the length field of an array object. He uses the native read function to disclose memory content beyond the array buffer, e.g., the vTable pointer of a consecutive object.

5.1.2 *vTable Hijacking*

To hijack the program's control flow, the adversary must overwrite a code pointer holding the destination of an indirect branch instruction. C++ virtual function tables (vTables) are commonly used for this purpose. The vTable is used to resolve virtual functions call targets at run time and contains an array of pointers to virtual functions, along with other meta-data. The entries of a vTable cannot be overwritten because they reside in read-only memory. However, each C++ object that uses virtual functions maintains a pointer to its corresponding vTable. Since this pointer is a field of the object, it is stored in writable memory. The adversary can exploit a memory corruption vulnerability to overwrite the vTable pointer of a C++ object with a pointer to a fake vTable which he created and injected beforehand. Instead of the original table of function pointers, all function pointers in the fake vTable will point to the code the adversary aims to leverage for a code-reuse attack. Lastly, after overwriting the vTable pointer of an object, the adversary uses JavaScript interfaces to the native object to invoke a virtual function from the fake vTable.

5.2 Proof-of-Concept Exploit

Our exploit performs the following steps: (i) Gain arbitrary read and write capabilities, (ii) locate the stack and disclosing its contents, and (iii) bypass the CFI check and hijack the control flow.

The re-introduced vulnerability (CVE-2014-3176) allows us to manipulate the data fields of JavaScript objects on the heap, such as ❶ in Figure 2. Once an array-like object has been corrupted, we can access adjacent memory location without failing a bounds check (see ❷ in Figure 2). In our exploit, we use the corrupted object to manipulate the buffer pointer field of a JavaScript `ArrayBuffer` instance. By setting the buffer pointer to the address we want to access, we can then read and write arbitrary memory by accessing the first element of the `ArrayBuffer` via the JavaScript interface. There are many ways to corrupt array-like objects, hence, our exploit does not depend on a specific type of vulnerability.

Disclosing Data Structures.

Chromium places different memory objects in different heaps. For instance, the array instance in Figure 2 is stored in the *object heap* while the data buffer it contains is in the *buffer heap*. The use of separate heaps prevents exploit techniques such as *heap feng shui* [46] which the adversary has used to co-locate vulnerable buffers and C++ objects [51].

However, during the analysis of Chromium's heap allocator, we found a way to force the allocator to place the vulnerable buffer at a constant offset to metadata that is used by the allocator to manage the different heaps. Chromium's heap allocator, *PartitionAlloc*, pre-allocates memory for a range of different buffer sizes. However, when memory for a buffer is requested that was not pre-allocated, *PartitionAlloc* will request memory from the operating system. Since *PartitionAlloc* needs to manage the dynamically allocated memory buffers, it requests two additional, consecutive memory pages from the operating system. The newly requested memory is organized as follows:

(i) Meta information of allocated memory. This includes a pointer to the main structures of *PartitionAlloc*, which contains all information to manage existing and future allocations.

(ii) Guard page. This page is mapped as inaccessible, hence, continuous memory reads/writes will be prevented. However, it does not prevent non-continuous reads/writes.

(iii) Memory to fulfill allocation request. This is the memory that is used by *PartitionAlloc* to allocate buffers.

By allocating a large buffer (e.g., 1MB) which is very unlikely to happen during normal execution, we ensure that *PartitionAlloc* will allocate a new structure as previously described. We further know that the requested buffer will be placed at the start of (3), because it is the first buffer of this size. Since the offset between (i) and (iii) is constant, we can disclose the pointer to the main meta-data structure of *PartitionAlloc*. This allows us to identify all memory addresses used by the heap allocator, as well as predict which memory addresses will be used for future allocations.

This is a very powerful technique as we can predict the memory address of every C++ object that is created. Further we can control which objects are created at run time via the JavaScript interface. Hence, it becomes very hard to hide information (e.g., a shadow stack address) because as long as any object contains a pointer to the hidden information, we can disclose the information by creating the object and disclosing its memory.

Finally, in our attack, we choose to allocate an object that contains a vTable pointer, i.e. the `XMLHttpRequest` object. By overwriting the vTable pointer of this object with a pointer to a fake vTable, we can hijack the control flow (see Section 5.1.2).

Disclosing the Stack Address.

To disclose and corrupt values on the stack to bypass CFI checks, we must first locate the stack in memory. In contrast to the heap, objects on the stack are only live until the function that created them returns. Hence, it is challenging to find a pointer to a valid stack address within the heap area. However, we noticed that Chromium's JavaScript engine V8 saves a stack pointer to its main structure when a JavaScript runtime library function is called. Since the `ArrayBuffer.read()` function, which we use for information disclosure, is part of the runtime library, we can reliably read a pointer that points to a predictable location on the stack. The remaining challenge is to find a reference to a V8 object, because V8 objects are placed on a different heap than Chromium's objects. Hence, we need to find a reference from an object whose address we already disclosed to the V8 object that stores the stack address. We chose `XMLHttpRequest`, because it contains a pointer to a chain of other objects which eventually contain a pointer to the V8 object. Once we disclose the address of this object, we can disclose the saved stack pointer.

At this point we have arbitrary read and write access to the memory and have disclosed all necessary addresses. Hence, we now focus on implementing the attacks described in Section 4.

5.2.1 *Bypassing IFCC*

IFCC implements fine-grained forward-edge CFI and is vulnerable to attacks that overwrite registers which are spilled on the stack. For brevity, we omit the bypass of VTV. However, from a conceptual point of view there is no difference between the IFCC bypass and the one for VTV. Tice et al. [50] assume that the stack is protected by StackGuard [13] which implements a canary for the stack to prevent any stack attacks. In practice, this does not prevent the adversary from overwriting the return address. Since IFCC focuses on the protection of CFG forward edges, we assume an ideal shadow stack to be in place that cannot be bypassed, though this might be hard to implement in practice.

IFCC protects indirect function calls by creating, at compile time, a list of functions that can be reached through indirect calls. It then creates a trampoline, i.e., a simple jump instruction to the function, for every function in this list. The array of all trampolines is called *jump table*. Finally, every indirect call is instrumented so it can only target a valid entry in the jump table.

Listing 1 contains the disassembly of an instrumented call. In the Line 8 and 9, the target address of the indirect call and the address of the jump table are loaded into registers. Subtracting the base address of the target pointer and then using a logical *and* is an efficient way of ensuring that an

```
1  F0:
2  call  F0_next
3  F0_next:
4  pop   ebx       ; load abs. address of F1
5  [...]
6  call  F_spill
7  [...]
8  mov   edi, [eax+4] ; load address F_target
9  mov   eax, [ebx-149C8h] ; load jump-table
10 mov   ecx, edi
11 sub   ecx, eax ; get offset in jump table
12 and   ecx, 1FFFF8h ; enforce bounds
13 add   ecx, eax ; add base addr jump table
14 cmp   ecx, edi ; compare target address
15 jnz   cfi_failure
16 call  edi       ; execute indirect call
17
18 F_spill:
19 push  ebx
20 [...]     ; overwrite of ebx happends here
21 pop   ebx
22 ret
```

Listing 1: Disassembly of an indirect call that is instrumented by IFCC.

offset within the jump table is used. Finally, this offset is added again to the base address of the jump table. This ensures that every indirect call uses the jump table, unless the adversary can manipulate the `ebx` register. As we explained in Section 4.1 `ebx` is a callee-saved register and therefore spilled on the stack during function calls.

For our exploit we target a protected, virtual function call F_{target} that is invoked (Line 16) *after* another function F_{spill} is called (Line 6), see Listing 1. During the execution of F_{spill} the `ebx` register is spilled on the stack (Line 19): we overwrite the target address of F_{target} through vTable injection (see Section 5.1.2) and the saved `ebx` register. We overwrite the saved `ebx` register such that Line 9 will load the address of our gadget. After F_{spill} finishes execution, the overwritten register is restored and used to verify the subsequent call in F_{target}. The check will pass and Line 16 will call our first gadget. After the initial bypass of CFI, we use unintended instructions to avoid further CFI checks.

Although 64-bit x86 offers more general purpose registers, our analysis of a 64-bit, IFCC-protected Chromium version exposed that around 120,000 out of 460,000 indirect calls CFI checks (around 26%) are vulnerable to our attacks. We did not manually verify if all of these CFI checks are vulnerable. However, for a successful attack it is sufficient that only one of these CFI checks is vulnerable to our attack. We exploited one vulnerable CFI check to implement a similar attack and bypass IFCC for the 64-bit version of Chromium.

5.2.2 *Bypassing fine-grained CFI*

It seems that overwriting a user-mode return address used by a system call is straightforward. However, we encountered some challenges during the implementation. The first challenge is being able to correctly time the system call and the overwrite of the return address. We found the most reliable way is to spawn two threads: one thread constantly makes the system call and the other constantly overwrites the return address. The attack succeeded in 100% of our tests without any noticeable time delay.

We can utilize the *Web Worker* HTML5 API [54] to create a dedicated victim thread. During our analysis to find a suitable function that eventually invokes a system call, we noticed that an idle thread is constantly calling the `ZwWaitForSingleObject` system call which is shown in Listing 2. Line 4 shows the call that pushes the return address on the stack that is later used by the kernel to return to user mode.

```
1  ntdll!ZwWaitForSingleObject:
2  mov  eax,187h    ; System call number
3  mov  edx,offset SystemCallStub
4  call [edx]       ; call KiFastSystemCall
5  ret  Ch
6
7  [...]
8
9  ntdll!KiFastSystemCall:
10 mov  edx,esp
11 sysenter
```

Listing 2: ZwWaitForSingleObject System Call on Windows 7 32-bit.

Another challenge is that the constant invocation of the system call might corrupt any ROP gadget chain we write on the stack. Hence, we overwrite the user-mode return address with the address of a gadget which sets the stack pointer to a stack address that is not constantly overwritten. From there on we use gadgets that are composed of unintended instructions [44] to bypass the instrumented calls and returns.

This exploitation technique can bypass any fine-grained CFI solution that aims to protect 32-bit applications on Windows.

6. MITIGATIONS

We consider possible mitigation techniques against our attacks. First, we describe our compiler patch for the IFCC/-VTV implementation vulnerability and measure its performance impact on the SPEC CPU2006 benchmarks. Subsequently, we discuss the broader problem of protecting the stack against memory disclosure and corruption attacks.

6.1 Patching IFCC

Recall that IFCC uses the base register containing the address of the GOT to reference the jump table validating the target of an indirect call (see Section 4.1). To prevent our attack presented in Section 5.2.1, we developed a compiler patch that safely reloads the GOT register before loading the CFI jump table. Our patch adds new instrumentation before the CFI check so this register is always re-calculated instead of being restored from the stack. With our proposed fix, IFCC uses three more instructions to validate each target which brings the total number of added instructions up to 15 per indirect call. Listing 3 shows an example of the IFCC instrumentation without our patch, and Listing 4 shows the reload we add on lines 12-17.

We measured the performance impact of this change using the SPEC CPU 2006 benchmark suite on a dual channel Intel Xeon E5-2660 server running Ubuntu 14.04 with Linux kernel 3.13.0. We selected only the benchmarks that have indirect calls since IFCC will not affect code that only uses direct calls. The benchmark results we report are medians over three runs using the reference inputs.

We report overheads relative to a baseline without IFCC enabled. Since IFCC uses link-time optimization, we also compile the baseline with link-time optimization turned on.

```
1     ; store current eip in ebx
2     call .next
3  .next:
4     pop ebx
5     add ebx, GLOBAL_OFFSET_TABLE
6     ...
7     ; call function which stores ebx to the
          stack
8     ...
9     ; Load destination function address
10    lea ecx, vtable+index
11    ; Load jump table entry relative to ebx
12    mov eax, [ebx + _jump_table_@GOT]
13    <perform CFI-check>
14    call ecx
```

Listing 3: Example IFCC assembly before fix

```
1     ; store current eip in ebx
2     call .next
3  .next:
4     pop ebx
5     add ebx, GLOBAL_OFFSET_TABLE
6     ...
7     ; call function which stores ebx to the
          stack
8     ...
9     ; Load destination function address
10    lea ecx, vtable+index
11
12    ; PATCH: Reload ebx with current eip,
          instead of
13    ;   untrusted, corruptible value
14    call .next2
15 .next2:
16    pop ebx
17    add ebx, GLOBAL_OFFSET_TABLE
18
19    ; Load jump table entry relative to ebx
20    mov eax, [ebx + _jump_table_@GOT]
21    <perform CFI-check>
22    call ecx
```

Listing 4: Example IFCC assembly after fix

Figure 3 shows that our patched version of IFCC performs between 0.12% and 1.19% slower (0.46% on average) than unpatched IFCC. Tice et al. [50] also found cases where IFCC outperforms the baseline, and we did not analyze these cases further. The patch for the 64-bit version is similar and was omitted for brevity.

We reported the weaknesses in IFCC and VTV and our mitigation for IFCC to the original developers of these mitigations.

6.2 Securing Stack

It is highly challenging to secure the machine stack against all types of attacks, since it must be readable and writable by the program. Similar to other exploit mitigation schemes, stack protection schemes can be categorized into schemes that rely on applying randomization to the stack or ensuring the integrity of the stack through isolation. We found that current stack randomization schemes introduce a lower performance overhead but remain vulnerable to our attack as the randomization secret can be disclosed, as we will discuss in the next section. On the other hand, isolating the stack can potentially mitigate our attacks. However, current

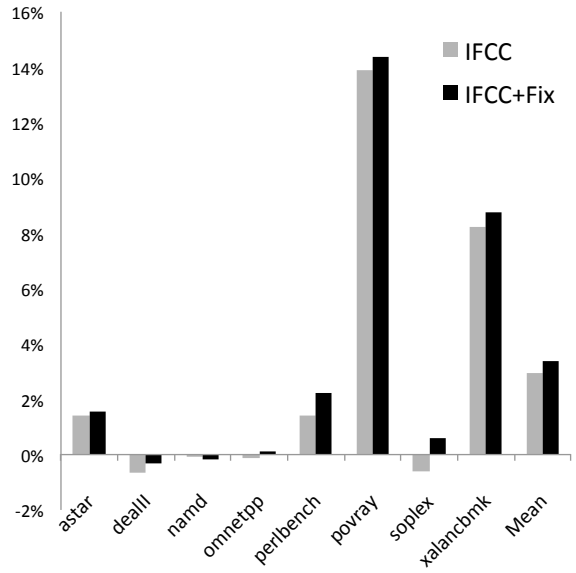

Figure 3: SPEC CPU2006 performance of IFCC-protected programs before and after we applied our fix relative to an unprotected baseline.

stack mitigation techniques are either not effective or suffer from non-negligible performance overheads.

Next, we shortly discuss the effectiveness of these mitigation schemes under our threat model.

6.2.1 *Randomization-based Defenses*

StackGuard [13] attempts to prevent stack-based buffer overflows by inserting a random stack cookie between potentially vulnerable buffers and the return address. However, this defense is insufficient against current attackers. An attacker with the capability to read the stack, as we have demonstrated with our attacks, can read and forge this cookie, even without an arbitrary memory write vulnerability.

The recently proposed StackArmor [11] further protects the stack not only against buffer overflows, but also stack buffer over/under reads and stack-based temporal vulnerabilities. However, StackArmor's protections are confined to the stack itself. Without any heap protection, an attacker can use heap-memory corruption to read and write arbitrary memory locations and can disclose metadata used by the StackArmor allocator to find and modify the stack.

6.2.2 *Isolation-based Defenses*

One possible mitigation strategy against our attacks is to isolate the stack from the regular data memory.

Lockdown [39] is a DBI-based (dynamic binary instrumentation) CFI implementation with a shadow stack. DBI rewrites the binary code of the application at run time, hence, it can control application memory accesses. This allows it to prevent access and leakage of the shadow stack address. However, these security guarantees come with an average run time overhead of 19% which is considered impractical.

Recently LLVM integrated a component of CPI, called SafeStack [32, 48]. It aims to isolate critical stack data, like return addresses or spilled registers from buffers that potentially can be overflown. During a static analysis phase

the compiler identifies buffers that are located on the stack and relocates them to a separate stack, called the *unsafe stack*. The regular stack is then assumed to be the *safe stack*. The separation of buffers and critical values is likely to prevent most stack-based memory vulnerabilities from being exploitable. However, if we can leak the stack pointer register (see Section 5.2), i.e., the pointer to the *safe stack*, we can overwrite the protected values.

Full CPI [32] provides more comprehensive protection of code pointers by isolating them from other memory objects. On 32-bit x86 the isolation is enforced through segmentation. In principle this can prevent our attacks, however, on 64-bit x86 or other architectures, e.g., ARM, this feature is not available. The authors suggest alternative implementations to the segmentation-based isolation. All come with their own pros and cons: While the randomization approach provides good performance, it was shown to be prone to information leakage attacks [20]. A more secure implementation is based on software fault isolation (SFI) [52], however, this adds an additional 7% [42] to the 8% average run-time overhead induced by CPI itself [32]. In general the overhead depends on the number of objects that must be protected, e.g., the authors report of CPI an overhead of 138% for a web server that serves dynamic webpages, which is impractical.

6.3 Securing CFI Implementations

Zeng et al. [56] compiled a list of requirements to implement a secure inline-reference monitor, in which they also mention the danger of stack-spilled variables. However, the threat of stack-spilled registers was not considered in two major compiler implementations. Our work proves that register spills are a severe threat to CFI, which should be address by future implementations.

Ultimately, while stack-oriented defenses help to mitigate stack vulnerabilities, they do not offer sufficient protection to complex software such as web browsers, where dynamic code generation, heap vulnerabilities and attacker-controlled scripting provide many alternative attack vectors to the adversary. Defenders must combine these types of defenses with other protection against heap-based memory corruption to be secure.

7. RELATED WORK

Memory disclosure poses a crucial threat to application security. Previous research on memory disclosure focused on leaking information about the code layout to bypass code randomization whereas we focus on data structures to identify memory locations that contain values that are critical for the enforcement of CFI.

Many exploit mitigations introduce randomness into the in-memory representation of applications. Such mitigations rely on the assumption that the adversary cannot read the memory. However, in the presence of memory disclosure vulnerabilities, assuming memory secrecy is neither justified nor realistic. We first discuss related offensive work on variety of memory disclosure vulnerabilities and bypasses of fine-grained CFI, and then devote our attention to defensive works that aim at resisting memory leakage.

7.1 Memory Disclosure Attacks

Bhatkar et al. [7] note that contemporary schemes (ASLR, StackGuard, PointGuard) are vulnerable if an adversary can read arbitrary values in memory. Strackx et al. [47] later demonstrated that memory disclosure through buffer overread errors allows attackers to bypass ASLR and stack canaries. Roglia et al. [22] then used return-oriented programming to disclose the randomized location of `libc`.

Observing that ASLR was highly vulnerable to memory disclosure, researchers argued that fine-grained code randomization solutions would provide sufficient resilience [19, 24, 27, 28, 33, 37, 53].

Snow et al. [45] introduced a novel type of memory disclosure attacks called just-in-time return-oriented programming (JIT-ROP) that is able to bypass not only ASLR but fine-grained code randomization as well. JIT-ROP exploits the design of memory paging in modern systems: it (i) identifies the page start and end of a leaked function pointer, (ii) disassembles the code page, and (iii) identifies code references on the disassembled page to repeat this process on other memory pages. Since JIT-ROP attacks must disassemble and analyze the disclosed code, it must be launched against scripting-capable victim applications such as web browsers or document viewers.

Bittau et al. [8] developed another memory disclosure attack against services that automatically restart after crashes. This attack exploits the fact that some servers (created using `fork` without `execve`) do not re-randomize after a crash. By sending such servers a malformed series of requests and by analyzing whether the requests cause the server to crash, hang, or respond, the adversary can guess the locations of the gadgets required to launch a simple ROP attack that sends the program binary to the remote adversary. Like JIT-ROP, this attack undermines fine-grained code randomization.

Siebert et al. [43] presented a memory disclosure attack against servers that uses a timing side-channel. By sending a malformed request to a web server, the adversary can control a byte pointer that controls the iteration count of a loop. This creates a correlation between the target of the pointer and the response time of the request that the adversary can use to (slowly) scan and disclose the memory layout of the victim process. In a similar vein, Hund et al. [29] exploit a timing side-channel to infer the memory of the privileged ASLR-randomized kernel address space.

Lastly, Evans et al. [20] were able to use memory disclosure to bypass an implementation of the code pointer integrity (CPI) defense by Kuznetsov et al. [32]. CPI works by storing control flow and bounds information in a "safe region" which is separate from non-sensitive data. This prevents control-flow hijacking and spatial memory corruption. Whereas the 32-bit x86 implementation uses memory segmentation to isolate the safe region, the fastest 64-bit x86 implementation uses information hiding. However, it turns out that the hidden safe region was sufficiently large to be located and parsed using a modified version of the memory disclosure attack by Siebert et al. [43]. Kuznetsov et al. also provide a 64-bit CPI implementation where the safe region is protected by SFI which has not been bypassed.

7.2 Attacks against fine-grained CFI

Carlini et al [9] provide evidence that static fine-grained CFI provides insufficient protection, and that a shadow stack is required to provide precise enforcement of the CFG. Furthermore, they demonstrate that CFI cannot defend against *non-control-data* attacks, where the control flow stays within the boundaries of the enforced CFG but the attacker mod-

ifies variables or function arguments such that the targeted application behaves maliciously.

CFI is only as effective as the CFG that is derived for the application. In *Control Jujutsu* Evans et al. [21], explore the limits of the state-of-the-art algorithm [34] that is used to derive forward edges in the CFG. They found that the derived CFG contains imprecisions that can be exploited and allow arbitrary code execution.

7.3 Preventing Memory Disclosure

Backes and Nürnberger [6] proposed Oxymoron, the first defense that aims at preventing JIT-ROP. Oxymoron prevents the step of the JIT-ROP attack in which it identifies references to other code pages. Oxymoron does so by making all references between code pages opaque using legacy x86 segmentation features. Davi et al. [17] show that Oxymoron can be bypassed in practice since JIT-ROP can be modified not to rely on following references between code pages by harvesting code pointers from C++ virtual tables instead.

Backes et al. [5] proposed an alternative defense against JIT-ROP called eXecute-no-Read (XnR). The goal of XnR is to improve upon DEP under which execute permissions imply read permissions. To emulate execute permissions without read permissions, XnR marks all but a small number of code pages as "not present". A modified page fault handler marks pages as executable (and readable) and simultaneously marks the last recently used executable page as "not present". As our experiments in Section 8 indicate, XnR in combination with function permutation does not provide protection against code-pointer leakage.

The HideM approach by Gionta et al. [23] implements execute-only memory via "TLB-desynchonization". This approach, which relies on pre-2008 hardware, directs read accesses to a different memory page than instruction fetches by the CPU. This avoids the small window of executable and readable pages. Nevertheless, HideM does not protect against code pointer leaks.

The Readactor approach by Crane et al. [14] implements execute-only memory using the second level address translation feature in modern processors with support for hardware accelerated virtualization. Redactor introduces code-pointer hiding, a technique which decouples all code pointers in attacker observable memory from the code layout. For instance, a return pointer does not point into a function, instead it points to a "return trampoline" which jumps to the original return address. Because the return trampoline is mapped with execute-only permissions, the adversary cannot read the original return address. In general, Readactor resists our attacks as it is resilient to code-pointer leakage. On the other hand, the trampoline addresses are still allocated on the stack and subject to StackDefiler. The adversary can collect trampoline addresses to identify call-preceded gadgets. However, it is not yet shown that the collected call-preceded gadgets are sufficient to mount a gadget-stitching attack [10, 18, 25, 26, 41].

8. DISCUSSION

Memory disclosure was previously used to attack code-randomization schemes [45]. Although attacking code randomization is not the main focus of this paper, it suggests itself to use stack disclosure against code randomization. In particular, we investigated the impact of stack disclosure against mitigation schemes that aim to prevent direct memory disclosure by marking the code segment as execute-only: XnR [5] and HideM [23]. We performed some preliminary experiments in which we used our capabilities to read the stack of a parallel thread to disclose a large number of return addresses. Considering that we can control which functions are executed in the parallel thread, we were able to leak the addresses of specific gadgets. The results of our experiments are that indirect code disclosure via return addresses can be used to bypass fine-grained code-randomization. In particular, we can bypass function permutation [31] or basic-block permutation [53] even when XnR or HideM are in place to protect against memory disclosure. Readactor by Crane et al. [14] performs code-pointer hiding and is not vulnerable to return address leakage. Further, the authors extended their work to protect function tables [15] which prevents vTable hijacking as described in Section 5.1.2.

9. CONCLUSION

We present StackDefiler a set of stack corruption attacks that we use to bypass CFI implementations. Our novel attack techniques corrupt the stack without the need for stack-based vulnerabilities. This contradicts the widely held belief that stack corruption is a solved problem. To the best of our knowledge, this paper presents the first comprehensive study of stack-based memory disclosure and possible mitigations.

Surprisingly, we find that fine-grained CFI implementations for the two premier open-source compilers (used to protect browsers), LLVM and GCC, are not safe from attacks against our stack attacks. IFCC spills critical pointers to the stack which we can exploit to bypass CFI checks. We verified that a similar vulnerability exists in VTV—a completely separate implementation of fine-grained CFI in a separate compiler. Next, we demonstrated that unprotected context switches between the user and kernel mode can lead to a bypass of CFI. Further, we show the challenges of implementing a secure and efficient shadow stack and provide evidence that information disclosure poses a severe threat to shadow stacks that are not protected through memory isolation. Finally, we analyzed several stack-based defenses and conclude they cannot counter our StackDefiler attack.

Based on our findings, we recommend that new defenses should (i) consider the threat of arbitrary memory reads and writes to properly secure a web browser and other attacker-scriptable programs, (ii) never trust values from writable memory, and (iii) recommend complementary approaches to protect the stack and heap to mitigate the threat of memory disclosure.

10. ACKNOWLEDGMENTS

The authors thank Andrei Homescu for providing insight into the V8 JavaScript engine. We thank Ferdinand Brasser and the anonymous reviewers for their suggestions and constructive feedback.

This work has been co-funded by the German Science Foundation as part of project S2 within the CRC 1119 CROSSING, the European Union's Seventh Framework Programme under grant agreement No. 609611, PRACTICE project and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC)

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA)

under contracts D11PC20024, N660001-1-2-4014, FA8750-15-C-0124, and FA8750-15-C-0085 as well as gifts from Google, Mozilla, Oracle, and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

Mauro Conti is supported by a European Marie Curie Fellowship (N. PCIG11-GA-2012-321980). This work is also partially supported by the Italian MIUR PRIN Project TENACE (N. 20103P34XC), and the University of Padua PRAT 2014 Project on Mobile Malware.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ICFEM'05, 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 2000.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[6] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[7] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, USENIX Sec, 2003.

[8] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[10] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[11] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*, NDSS, 2015.

[12] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium*, NDSS, 2014.

[13] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *8st USENIX Security Symposium*, USENIX Sec, 1998.

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[15] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRAP: Table randomization and protection against function reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[16] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2015.

[17] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS, 2015.

[18] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[19] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Smiecurity*, ASIACCS, 2013.

[20] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[22] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*, ACSAC, 2009.

[23] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, CODASPY, 2015.

[24] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, USENIX Sec, 2012.

[25] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[26] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[27] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: where'd my gadgets go? In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[28] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2013.

[29] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[30] Intel. Intel 64 and IA-32 architectures software de-

veloper's manual, combined volumes 3A, 3B, and 3C: System programming guide. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, 2013.

[31] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*, ACSAC, 2006.

[32] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[33] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[34] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2007.

[35] Microsoft. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[36] Microsoft. Control flow guard. https://msdn.microsoft.com/en-us/library/Dn919635.aspx, 2015.

[37] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[38] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

[39] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, DIMVA, 2015.

[40] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[41] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *17th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID, 2014.

[42] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *19th USENIX Conference on Security*, USENIX Sec, 2010.

[43] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[44] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[46] A. Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, BH US, 2007.

[47] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, EUROSEC, 2009.

[48] The Clang Team. Clang 3.8 documentation SafeStack. http://clang.llvm.org/docs/SafeStack.html, 2015.

[49] C. Tice. Improving function pointer security for virtual method dispatches. In *GNU Tools Cauldron Workshop*, 2012.

[50] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[51] VUPEN Security. Advanced exploitation of internet explorer heap overflow (pwn2own 2012 exploit). http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php, 2012.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, SOSP, 1993.

[53] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2012.

[54] Web Hypertext Application Technology Working Group (WHATWG). Chapter 10 - Web workers, 2015.

[55] Z. Yunhai. Bypass control flow guard comprehensively. In *Black Hat*, BH US, 2015.

[56] B. Zeng, G. Tan, and U. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

[57] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[58] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.