It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks

Stephen Crane*, Stijn Volckaert†, Felix Schuster‡, Christopher Liebchen§, Per Larsen*, Lucas Davi§, Ahmad-Reza Sadeghi§, Thorsten Holz‡, Bjorn De Sutter†, Michael Franz*
*University of California, Irvine
†Universiteit Gent, Belgium
‡HGI, Ruhr-Universität Bochum, Germany
§CASED, Technische Universität Darmstadt, Germany

ABSTRACT

Code-reuse attacks continue to evolve and remain a severe threat to modern software. Recent research has proposed a variety of defenses with differing security, efficiency, and practicality characteristics. Whereas the majority of these solutions focus on specific code-reuse attack variants such as return-oriented programming (ROP), other attack variants that reuse whole functions, such as the classic return-into-libc, have received much less attention. Mitigating function-level code reuse is highly challenging because one needs to distinguish a legitimate call to a function from an illegitimate one. In fact, the recent *counterfeit object-oriented programming* (COOP) attack demonstrated that the majority of code-reuse defenses can be bypassed by reusing dynamically bound functions, i.e., functions that are accessed through global offset tables and virtual function tables, respectively.

In this paper, we first significantly improve and simplify the COOP attack. Based on a strong adversarial model, we then present the design and implementation of a comprehensive code-reuse defense which is resilient against reuse of dynamically-bound functions. In particular, we introduce two novel defense techniques: (i) a practical technique to randomize the layout of tables containing code pointers resilient to memory disclosure and (ii) booby trap insertion to mitigate the threat of brute-force attacks iterating over the randomized tables. Booby traps serve the dual purpose of preventing fault-analysis side channels and ensuring that each table has sufficiently many possible permutations. Our detailed evaluation demonstrates that our approach is secure, effective, and practical. We prevent realistic, COOP-style attacks against the Chromium web browser and report an average overhead of 1.1% on the SPEC CPU2006 benchmarks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Code generation*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCS'15, October 12–16, 2015, Denver, Colorado, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813682.

General Terms

Security; Languages; Performance

Keywords

Code reuse; Exploits; Mitigations; COOP; C++; Compilers; Diversity; Randomization

1. MOTIVATION

Memory corruption vulnerabilities due to incorrect memory management have plagued software written in low-level languages for more than three decades. Many defenses have been proposed in the literature, but few meet the requirements for industry adoption [37]. Data Execution Prevention (DEP) is one of the few successful defenses that has seen widespread adoption. As a result, adversaries quickly shifted from injecting malicious code to reusing legitimate code. Code-reuse attacks, such as return-into-libc (RILC) [26] or return-oriented programming (ROP) [29], are far harder to mitigate with reasonable cost, and remain an active research topic.

One effective and efficient line of defense relies on randomization. The widespread address space layout randomization (ASLR) technique, for instance, randomizes the base addresses of various memory regions. However, adversaries can often exploit a memory corruption vulnerability to disclose the code layout, and rewrite the code-reuse payload accordingly. Initially, the problem seemed to be the coarse-grained nature of ASLR. This motivated many finer-grained code randomization schemes (systematized by Larsen et al. [22]). However, Snow et al. [35] demonstrated *just-in-time* code reuse (JIT-ROP) that exploits memory disclosure and malicious scripting to read the randomized code layout and construct a compatible code-reuse payload on the fly. Just-in-time code reuse can be countered by preventing the adversary from directly reading the code layout [3, 17]. However, preventing reading of code pages with execute-only memory is insufficient, because the adversary can disclose the code layout indirectly by reading code pointers on data pages. Crane et al. [11] presented *Readactor*, a security framework against all types of JIT-ROP attacks. Their solution allows for hiding code pointers and just-in-time compiled code from adversaries. However, very recently Schuster et al. [31] describe a new type of attack called *counterfeit object-oriented programming* (COOP), which reuses whole functions by abusing the C++ virtual dispatch mechanism. Targets of C++ virtual function calls and calls to dynamically linked functions are

looked up in various tables; they are thus also referred to as *dynamically-bound* functions.

At a high level, COOP attacks are conceptually similar to return-into-libc (RILC) attacks, but reuse virtual functions rather than functions from procedure linkage tables. These *function reuse attacks* remain unaffected by sophisticated JIT-ROP defenses such as Readactor [11] because these defenses do not introduce randomness into the various dynamic dispatch mechanisms found in today's software. Moreover, function reuse attacks are particularly hard to mitigate because legitimate function calls are hard to distinguish from malicious ones.

Goals and contributions: Although many exploit mitigations can be bypassed [6, 30, 31, 35], more advanced defense techniques are set to enter practice in the near future. Prominent examples of these security enhancing mechanisms are *control-flow integrity* [1], *code-pointer integrity* [21], and *fine-grained code randomization* [22]. While these defenses raise the bar to exploitation relative to current countermeasures, they differ with respect to performance, practicality, and security. Moreover, the COOP attack bypasses all published binary-only control-flow integrity (CFI) systems. In light of this attack and since software diversity is known to be efficient [14, 19], practical, and comprehensive [11], the goal of this paper is to tackle the gaps in existing randomization approaches by preventing function reuse attacks such as COOP and RILC. Specifically, we extend state-of-the-art code randomization defenses by (i) permuting tables containing code pointers while preserving program semantics, (ii) inserting *booby traps* into these tables to mitigate probing attacks, and (iii) transforming tables so we can use execute-only page permissions to prevent memory disclosure. Our main contributions are:

- **Resilience to Function-Reuse Attacks** We present Readactor++, the first probabilistic defense against function reuse attacks such as COOP and RILC that are not mitigated by existing code randomization techniques. We designed our defense under the challenging and realistic assumption that an adversary can read and write arbitrary memory.
- **Novel Techniques** We introduce compile-time and load-time transformations that work in concert to transform and randomize virtual function tables and dynamic linker tables.
- **Realistic and Extensive evaluation** We provide a full-fledged prototype of Readactor++ that protects applications against function reuse attacks and present the result of our detailed evaluation. We report an additional average overhead of 1.1% on the compute-intensive SPEC CPU2006 benchmarks over existing execute-only systems. We also show that Readactor++ scales to programs as complex as Google's popular Chromium web browser.

2. TECHNICAL BACKGROUND

To provide the necessary background for the subsequent discussion, we briefly review the relevant implementation aspects of function dispatch and offensive techniques that reuse whole functions.

2.1 Virtual Function Calls

Object-oriented languages such as C++ and Objective-C support polymorphism; the target of a function call depends on the type of the object that is used to make the call. C++ supports object orientation based on classes, which are simply user-defined data types with fields and functions (also called *methods*). Classes can be extended through inheritance of methods and fields. In contrast to many modern languages, C++ supports multiple inheritance.

The C++ compiler generates different call mechanisms for invoking a receiving object's method depending on whether the callee is a virtual function. A subclass can override functions marked as virtual in its base class(es). Non-virtual calls are simply compiled to a static, direct call to the corresponding function. For virtual function calls however, the exact callee depends on the type of the object that receives the call at runtime. Therefore, at virtual function call sites, the program first retrieves a pointer to a virtual function table (*vtable*) from the receiver object and then indexes into this table to retrieve a function pointer to the correct callee. As explained in Section 2.3, adversaries abuse indirect calls to virtual functions by corrupting objects to point to a vtable of their choice, thus controlling the call destination.

2.2 Dynamic Linking

Dynamic linking allows programmers to group functions and classes with related functionality into a single module. Programs can share dynamically linked modules, which simplifies maintenance and reduces code duplication both on disk and in memory.

Symbols are the basic building blocks of dynamic linking. Each module defines a set of symbols that it exposes to other modules through a symbol table. These symbols typically correspond to exported functions and variables. A module can refer indirectly to symbols which are not defined within the module itself. Symbol addresses in the table are resolved at run time by the dynamic linker. Many binary file formats support *lazy binding*. With lazy binding, the addresses of external symbols are resolved when they are first used. Lazy binding can improve the startup time of some applications, which is why some binary formats enable it by default.

The tables that support dynamic linking are an ideal target for attackers. Computed symbol addresses for instance are usually kept in tables in writable (and readable) memory. Furthermore, to support lazy binding, the meta-data necessary to resolve a specific symbol's address is also kept in readable memory at all times. These sources of information can be of use in an information leakage attack.

In ELF binaries, several specific dynamic linking tables are prone to information leakage. Attackers can collect function addresses from the Global Offset Table (GOT). The GOT stores the addresses of global and static elements of the program, including the addresses of all imported functions for use in the Procedure Linkage Table (PLT). The PLT on the other hand contains a set of trampolines that each correspond to a code pointer in the GOT. An attacker can infer the layout of both the GOT and the PLT tables from the relocation, symbol and string tables in the binary.

2.3 Exploitation and Code Reuse

Memory corruption is a well-known way to exploit vulnerabilities in programs written in unsafe languages [37]. A buffer overflow is a type of spatial memory corruption whereas use-after-free is a type of temporal memory corruption. Adversaries use memory corruption to inject code or

data, to read memory contents, and to hijack the execution by overwriting control-flow data.

Because DEP prevents code injection in most programs, code-reuse attacks are most common. Initially, adversaries reused code from linked libraries such as `libc` [26]. Therefore, this class of attacks is called return-into-libc (RILC), even when the reused functions are from another library. The simplest RILC attack uses memory corruption to (i) prepare for a function call by writing the arguments on the stack and (ii) redirect the control flow to a dynamically linked function such as `system()` in `libc`. In case the adversary wants to perform more than one call to a library function, the stack needs to be prepared for the next call. To do so, adversaries reuse short instruction sequences inside functions, called *gadgets*. Attacks that chain together gadgets whose last instruction is a return (or another free branch [7, 8]) are now known as ROP attacks [29].

Conventional ROP attacks are stack-oriented because they require the return instruction to read the next gadget address from the corrupted stack [29]. Since stack-based vulnerabilities are rare, attackers shifted to heap-oriented ROP. However, these attacks require a heap-based vulnerability along with a stack pivot sequence which sets the stack pointer to the start of the injected return-oriented payload on the heap. The former is typically achieved by vtable hijacking attacks [28]. As described above, a vtable holds pointers to virtual functions. While the function pointers are stored in read-only memory, the vtable pointer itself is stored in writable memory. Hence, an adversary can inject a malicious vtable, overwrite the original vtable pointer, write the return-oriented payload, and wait until the next indirect call dereferences the vtable pointer to invoke a virtual function. The correct virtual function is not called, due to the vtable pointer overwrite, but instead the code referenced by the previously injected malicious vtable is executed. In many exploits, the first invoked sequence is the aforementioned stack pivot, which sets the stack pointer to the start of the injected return-oriented payload.

The more recent COOP code-reuse technique described by Schuster et al. [31] starts by hijacking the control flow, as in other code-reuse attacks. Instead of directing the control flow to a chain of ROP gadgets, COOP attacks invoke a sequence of virtual function calls on a set of counterfeit C++ objects. This carefully crafted exploit payload is injected in bulk into a memory region that the attacker can access. Notice that constructing this payload requires knowledge of the exact layout of the relevant objects and vtables.

Whereas ROP relies on return instructions to transfer control from one gadget to another, COOP uses a so-called *main loop* gadget (or *ML-G*), which is defined as follows: "A virtual function that iterates over a container [...] of pointers to C++ objects and invokes a virtual function on each of these objects" [31]. In addition, there are certain platform-specific requirements that a virtual function must meet in order to be usable as an ML-G. On x86-64, for example, the calling convention specifies a set of registers that are used for passing explicit arguments to C++ functions and which should not be overwritten within the ML-G's loop. The ML-G is the first virtual function that is executed in a COOP attack and its role is to dispatch to the other virtual functions, called *vfgadgets*, that make up the COOP attack. Similar to ROP gadgets, vfgadgets fall into different categories such as those that perform arithmetic and logic operations, read and write

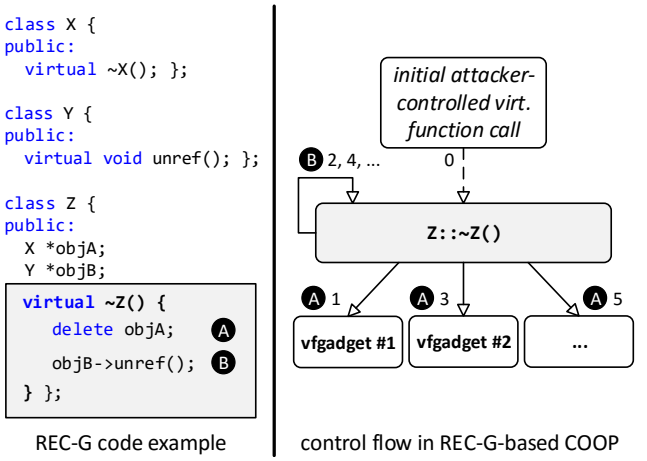

Figure 1: Left: C++ code example of a REC-G (`Z::~Z()`); right: the corresponding adversary-induced control flow; arrows indicate a *calls* relationship and are numbered in the order of their execution. The labels *A* and *B* refer to the call sites in `Z::~Z()` respectively.

to memory, manipulate the stack and registers, etc. We refer to Schuster et al. for a full treatment of COOP [31].

3. EXTENDING COOP

It is natural to ask whether COOP attacks can be mitigated by eliminating potential ML-Gs in an application. To disprove this hypothesis, we developed two refined versions of COOP that do not require ML-Gs and emulate the original *main loop* through *recursion* and *loop unrolling*. For brevity, we only discuss the x86-64 platform in this section. However, the described concepts directly extend to other platforms such as x86-32.

Recursive COOP. In general, all computation expressed through iteration can also be expressed via recursion. This also applies to COOP's main loop. We identified a common C++ code pattern that can be used to emulate an ML-G using recursion without compromising the expressiveness of COOP. We refer to a virtual function that follows this pattern as a *REC-G* (*recursion vfgadget*). To understand how a REC-G works, consider the C++ code in Figure 1: `Z::~Z()` is a typical (virtual) destructor. It deletes the object `objA` and removes a reference to `objB`. Consequently, a virtual function is invoked on both objects `objA` and `objB`. In case `Z::~Z()` is invoked on an adversary-controlled *counterfeit object*, the adversary effectively controls the pointers `*objA` and `*objB` and he can make these pointers point to injected counterfeit objects.

Accordingly, `Z::~Z()`, and REC-Gs in general, allow the adversary to make *two* consecutive COOP-style vfgadget invocations; we refer to these as invocations Ⓐ and Ⓑ. However, by using Ⓑ to repeatedly invoke the REC-G itself again, the adversary can invoke a new vfgadgets via Ⓐ with each recursion. This effectively enables the adversary to invoke an *arbitrary* number of vfgadgets given a REC-G. The right side of Figure 1 schematically depicts the adversary-induced control flow in a REC-G-based COOP attack: actual vfgadgets are invoked via Ⓐ, whereas Ⓑ is used to branch recursively to the REC-G (here `Z::~Z()`). Note how any compiler-

generated x86-64 assembly implementation of `Z::~Z()` is unlikely to touch argument registers: `Y::unref()` does not take any arguments and, by definition, the same applies to the destructors `X::~X()` and `Z::~Z()`. Accordingly, for the REC-G `Z::~Z()`, the adversary can freely use these registers to pass arguments from one vfgadget to another (all invoked via call site Ⓐ) as described by Schuster et al. [31]. We stress that not only destructors but any virtual function that follows the pattern of `Z::~Z()` and consecutively invokes virtual functions on (at least) two distinct and adversary-controlled object pointers may be used as a REC-G.

Intuitively, REC-Gs should generally be prevalent in C++ applications. Indeed, by applying basic pattern matching on x86-64 assembly code, we were able to identify a range of easy-to-use REC-Gs in different common C++ libraries and popular applications. For example, Figure 8 in the Appendix shows the C++ code of a side-effect free REC-G in the Boost C++ library collection; figures 9, 10, and 11 depict similar REC-Gs in the Qt C++ framework, Microsoft's Visual C++ runtime, and the Chromium web browser respectively.

Unrolled COOP. Given a virtual function with not only *two* consecutive virtual function invocations (like a REC-G) but *many*, it is also possible to mount a simple *unrolled* COOP attack that does not rely on a loop or recursion. Consider for example the following virtual function:

```
void C::func() {
 delete obj0; delete obj1; delete obj2; delete obj3;}
```

If objects `obj0` through `obj3` each feature a virtual destructor, `C::func()` can be misused to consecutively invoke four vfgadgets. We refer to virtual functions that enable unrolled COOP as UNR-Gs. Observe that, similar to `Z::~Z()`, a compiler-generated x86-64 assembly implementation of `C::func()` is unlikely to touch any argument registers. This again enables the adversary to freely pass arguments from one vfgadget to another.

We found that even long UNR-Gs are not uncommon in larger C++ applications. For example, in recent Chromium versions, the virtual destructor of the class `SVGPatternElement` is an UNR-G allowing for as many as 13 consecutive virtual function invocations. In practice, much shorter UNR-Gs are already sufficient to compromise a system; we demonstrate in Section 7 that the execution of three vfgadgets is sufficient for an adversary to execute arbitrary code.

4. ADVERSARY MODEL

We consider a powerful, yet realistic adversary model that is consistent with previous work on code-reuse attacks and mitigations [11, 13, 24, 31, 35]. We rely on several existing and complementary mitigations for comprehensive coverage.

Adversarial Capabilities.

- **System Configuration:** The adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application.
- **Vulnerability:** The target application suffers from a memory corruption vulnerability that allows the adversary to corrupt memory objects. We assume that the attacker can exploit this flaw to read from and write to arbitrary memory addresses.
- **Scripting Environment:** The attacker can exploit a scripting environment to process memory disclosure information at run time, adjust the attack payload, and subsequently launch a code-reuse attack.

Defensive Requirements.

- **Writable ⊕ Executable Memory:** The target system ensures that memory can be either writable or executable, but not both. This prevents an attacker from either injecting new code or modifying existing executable code.
- **Execute-only Memory:** We build on previous systems which enforce execute-only memory pages, i.e., the CPU can fetch instructions but normal read or write accesses trigger an access violation. See Section 5.2 for further discussion of this component.
- **JIT Protection:** We assume mitigations are in place to prevent code injection into the JIT code cache and prevent reuse of JIT compiled code [11, 18, 36]. These protections are orthogonal to Readactor++.
- **Brute-forcing Mitigation:** We require that the protected software does not automatically restart after hitting a booby trap which terminates execution. In the browser context, this may be accomplished by displaying a warning message to the user and closing the offending process.

5. Readactor++

We start by giving a conceptual overview of our approach, Readactor++, and then discuss each of its major components.

5.1 Overview

Our goal is to show that probabilistic defenses can thwart function-reuse attacks. When combined with memory leakage resilient code randomization, we can protect against the full range of known code-reuse attacks.

Unlike ROP attacks, which reuse short instruction sequences, COOP and RILC reuse dynamically-bound functions called through code pointers in read-only memory. To construct a COOP payload, the adversary must know (or disclose) the exact representations of objects and vtables. Similarly, RILC attacks invoke functions through the PLT which requires knowledge of the layout of this table. Our key insight is that we can permute and hide the contents of these tables even from an adversary that can disclose arbitrary readable memory.

Figure 2 shows a system overview. We use a staged randomization approach [5, 24] in which binaries are instrumented during compilation so that they randomize themselves when loaded into memory. Our compiler maintains all necessary information for load-time randomization and stores this metadata in the binary for use at load time. Since we gather this information during compilation, our prototype does not need to perform static analysis at load time. Note that although our prototype uses a compiler, our approach is compatible with binary rewriting as long as the vtable hierarchy and all virtual and PLT call sites can be recovered through static analyis, as in VTint [40].

During compilation (left side of Figure 2), we first ensure that code can be mapped in execute-only memory and perform code-pointer hiding to prevent function pointers and return addresses from disclosing the code layout [11]. Most importantly, we use a novel transformation step to make vtables compatible with execute-only memory and protect function pointers from disclosure. We split virtual tables into

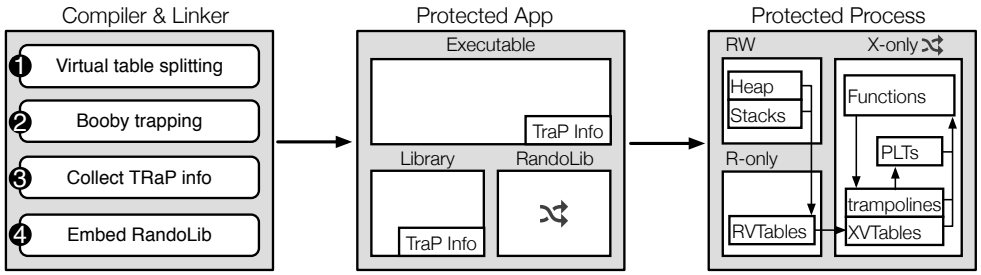

Figure 2: System overview. A specialized compiler (left) creates Readactor++ apps (middle) that randomize their in-memory representation (right). A small runtime component, `RandoLib`, uses TRaP meta-data embedded in binaries to safely permute the layout of vtables and procedure linkage tables without the need to disassemble the entire application. We prevent disclosure of randomized code using execute-only memory.

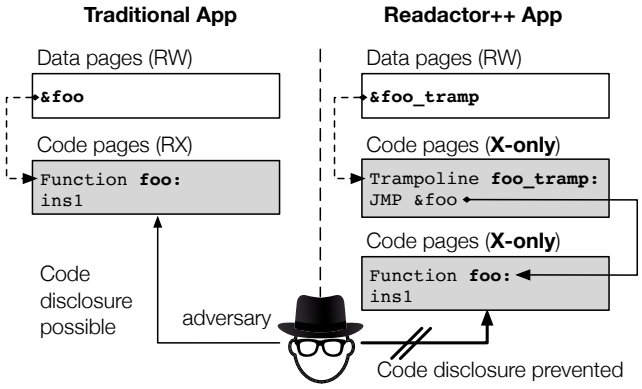

Figure 3: In traditional apps, adversaries can locate and read functions by following code pointers (left). We only store pointers to trampolines in readable memory (right). Trampolines prevent indirect disclosure of the function layout because trampolines are non-readable and their layout is randomized independently of other code.

a read-only part (rvtable) and an execute-only part (xvtable), which we randomize at load time. Section 5.3 elaborates on this step.

Second, we insert booby trap entries [10] in the xvtables. We also insert booby traps into the PLT during linking. Booby traps are code snippets that lie dormant during normal program execution and terminate the program and alert the host system if they are ever invoked. Booby traps prevent adversaries from randomly executing the vtable or PLT entries to indirectly disclose the table layout; Section 5.5 provides the details.

The third compilation step collects meta-data which we call "TRanslation and Protection" (TRaP) information. We embed TRaP information in the output binaries to support rapid load-time randomization. We link our runtime randomization engine, `RandoLib`, into the output binary in the final and fourth step.

The middle third of Figure 2 shows the contents of a protected binary. The main executable and each program library contains TRaP information that is read by `RandoLib` and used to randomize the vtables and PLT at load time. We elaborate on our vtable and PLT randomization techniques in Section 5.3 and 5.4, respectively.

The right side of Figure 2 gives an overview of the in-memory representation of applications protected by Readactor++. Global variables, the program stacks, and the heap are stored in readable and writable memory. All code areas are subject to fine-grained randomization and are protected against direct and indirect disclosure of the code layout; Section 5.2 provides additional details. Vtables, which are normally stored in read-only memory, are split into a read-only part (rvtable) and an execute-only part (xvtable) which is randomized. Because rvtables can be read by attackers, they do not store any code pointers directly. Instead, rvtables contain a pointer to their corresponding xvtable. Each entry in an xvtable is a direct jump rather than a code pointer so the table can be stored in execute-only memory to prevent adversaries from reading its contents.

5.2 Countering Memory Disclosure

Attacks against web browsers and other software that host scripting engines are particularly powerful because the combination of adversarial scripting and memory corruption can often allow adversaries to access arbitrary memory pages. In general, we distinguish between *direct* memory disclosure, where the adversary read code pages as data [35], and *indirect* memory disclosure, where the adversary reads code pointers stored in program vtables, stacks, and heaps to learn the code layout while avoiding direct accesses to code pages [13]. Execute-only schemes such as XnR [3] and HideM [17] prevent direct memory disclosure. Readactor [11] offers additional protection against indirect disclosure of code through code pointer harvesting. To prevent indirect disclosure, the Readactor system uses a technique called code-pointer hiding. They replace all code pointers stored into memory with pointers to trampolines instead as shown in Figure 3. Trampolines are simply direct jumps with the same target as the code pointer it replaces. Because trampolines are code stored in execute-only memory, adversaries cannot "dereference" trampolines to disclose the code layout. Execute-only memory and code-pointer hiding provides the foundation that we build upon when randomizing tables containing code pointers.

5.3 Vtable Randomization

C++ objects contain a hidden member, `vptr`, which points to a vtable. We could randomize the layout of objects to make the `vptr` harder to locate. However, C++ objects are stored on the heap and stack which are necessarily readable, so an adversary can use a memory disclosure vulnerability

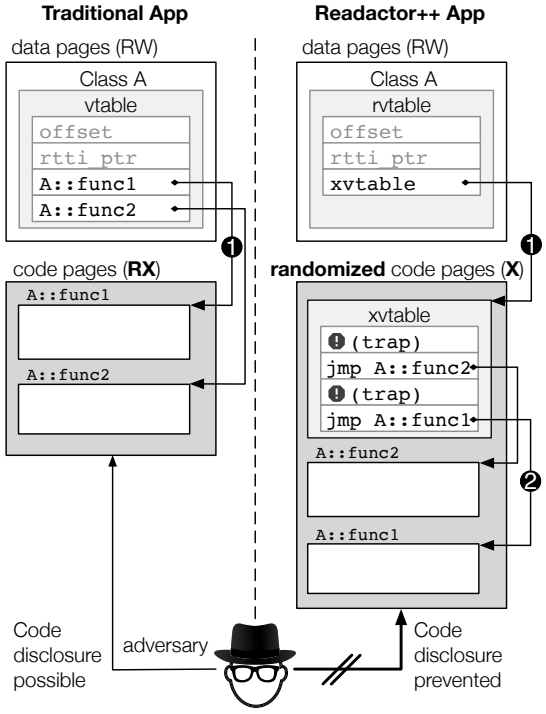

Figure 4: Without protection, an adversary that can read arbitrary memory can read the vtable contents and disclose the function layout. In Readactor++ apps, the readable part of vtables (rvtable) contain no code pointers and the executable and randomized part (xvtable) is not readable.

to parse the heap and discover how objects of a given class are randomized. We therefore chose to randomize the layout of vtables instead.

If we only randomize the ordering of vtable entries, an adversary can still follow `vptr`'s in the heap or stack to disclose the vtable layout. Legitimate vtables are most commonly stored on pages with read permissions. To prevent disclosure of the vtable layout after randomization, we want to prevent adversaries from reading the part of vtables that contains code pointers. Our solution is to transform read-only code pointers into execute-only code. We encode each code pointer p as a direct jump using the value of p as the immediate operand. In addition to code pointers, vtables contain other data such as Run-Time Type Information (RTTI). We therefore split each vtable into two parts: a read-only part called the *rvtable* and an execute-only part called the *xvtable*. We can either place the rvtable and xvtable on successive memory pages or we can add a pointer from the rvtable to the xvtable. If we choose the former approach, we need to pad rvtables to the nearest page boundary which wastes memory. We instead add a new field to the rvtable, `xpointer`, referencing the corresponding xvtable. Figure 4 shows how a traditional vtable (left) is split into an rvtable and an xvtable (right).

After splitting the vtable and inserting booby trap entries, we can securely randomize the ordering of each class's virtual functions. Since we store the xvtable in execute-only memory, an attacker cannot de-randomize the ordering of functions in the xvtable. We randomly permute the ordering of each class's virtual functions and rewrite all vtables in a semantics preserving way; Section 6.2 describes how we do this. After randomizing the function ordering in the vtables, we must rewrite all virtual call sites in the program to use the new, permuted ordering. These virtual calls use a static offset from the start of a vtable to dispatch to the correct function. We rewrite this offset to correspond to the new, randomized index of the callee. As Readactor++ extends Readactor, all virtual call sites are also augmented with trampolines that hide actual return addresses at run time.

5.4 Procedure Linkage Table Randomization

A call to a dynamically linked function goes though the PLT of the calling module. Each entry in the PLT is a code sequence that (i) loads a function pointer from the GOT into a register and (ii) transfers control to the target function through an indirect jump. To enable lazy binding, each function's default address in the GOT initially points to a function that resolves the external function address. The left side of Figure 5 shows the lazy binding process.

RILC attacks must know or discover the layout of the PLT to succeed. For ASLR'ed Linux applications, PLT entries are laid out contiguously and only the base address is randomized. One way to improve security is to eliminate the PLT altogether. However, this approach is problematic because the PLT or similar tables on Windows are essential to support dynamically linked functions.

As shown on the right side of Figure 5, we chose to randomize the order of PLT entries, insert booby traps, and store the result in execute-only memory. If we had only done this, an adversary could still read GOT or relocation information to disclose the ordering of PLT entries indirectly. We therefore switch from lazy to eager binding resolution of PLT entries. This allows us to discard the GOT. Our specialized compiler determines which calls will be routed through the PLT and stores the location of each such call site in the TRaP section of the output ELF file. At load time, our runtime randomization engine, `RandoLib`, converts all pointers in the readable GOT into execute-only trampolines in the PLT. To do so, we precompute all of the function addresses in the GOT. Then, we rewrite all the trampolines in the PLT so they jump to the target function directly rather than reading the function address from the GOT and indirectly jumping to it. This allows us to remove the code pointers from the GOT altogether. By stripping the GOT from all code pointers, we can also remove its associated relocation information from the memory. We then shuffle the trampolines in the PLT. This transformation prevents attackers from inferring the layout of the PLT or leaking the code pointers it contains. Finally, we use the TRaP information stored by our compiler to rewrite all call sites that point into the PLT with the corresponding randomized address.

5.5 Countering Guessing Attacks

Since we prevent attackers from directly reading valuable tables such as the PLT and vtables, we expect that attackers may try to execute xvtable entries and other addresses in execute-only memory to guess their contents. Researchers have shown that brute-force attacks can bypass diversity, especially in services that automatically restart without re-randomization after crashing [6, 15, 33, 34]. We use software booby traps to counter this threat [10]. The idea is that booby traps lie dormant during normal program operation but are likely triggered by adversarial probing. Booby traps therefore terminate the program immediately and notify an administrator or security software of the issue. Terminating

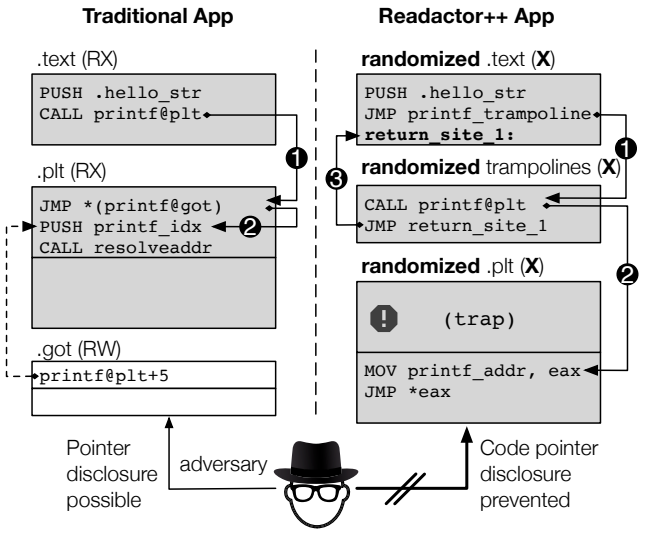

Figure 5: In traditional apps, functions call PLT entries directly (left). In Readactor++ apps, functions first jump to a trampoline which performs the actual call, so that actual return addresses are not leaked. Moreover, we resolve the targets of PLT entries, remove the GOT, and add booby traps to deter probing.

the program causes it to re-randomize as it starts up again, invalidating whatever knowledge the adversary has collected so far. When randomizing xvtables and procedure linkage tables, we insert booby traps. These are simply direct jumps to a booby trap handling routine. This probabilistically bounds the number of undetected guesses.

We take special care to ensure that there are many possible permutations for each table. Specifically, we ensure that each permuted table contains at least $n_{min} = 16$ entries. As explained Section 6.2, portions of each xvtable, which we call *sub-xvtables*, are permuted together. However we cannot alter the relative ordering of sub-xvtables. Thus, we ensure that each *sub*-xvtable has at least $n_{min} = 16$ entries. We also require that at least $k = \frac{1}{4}$ of the entries of each table are booby traps, so guessing the table entries by executing them will quickly lead to detection. Given a class with n virtual functions, we add $\max(\lceil k \cdot n \rceil, n_{min} - n)$ booby traps to its xvtable to meet these requirements. Both n_{min} and k are fully tunable for increased security. See Section 7.1 for a discussion of the security impact of these parameters.

6. IMPLEMENTATION

Our prototype implementation targets x86-64 Linux applications compiled with the LLVM/Clang compiler. However, our techniques are fully generalizable to other combinations of operating systems, compilers, and microarchitectures.

We prepare applications for load-time randomization using a modified compiler which we describe in Section 6.1. We then link the runtime randomization engine, described in Section 6.2, into the resulting binary. When the binary is loaded, the randomization engine uses the embedded TRaP information to randomize the host binary without the need for disassembly or complex binary analysis to recover the necessary information. Protected binaries are as simple to distribute and patch as current binaries and preserve compatibility with virus scanners, code signing, and whitelisting. A system that hosts Readactor++ binaries must support execute-only memory permissions; we build our implementation of Readactor++ on top of the Readactor system [11] in order to also mitigate indirect code disclosure.

6.1 Compiler Support

We modified the LLVM compiler to (i) generate code that can be mapped with execute-only permissions, (ii) split vtables into read-only and execute-only parts, and (iii) collect TRaP information to facilitate randomization of vtables and linker tables at load time.

Code and Data Separation As with other systems which require execute-only code pages, we must ensure that legitimate code never tries to read itself. We modify the LLVM compiler infrastructure and the Gold linker to ensure that the compiler will never mix code and readable data.

Splitting Vtables As we discuss in Section 5.3, we must split vtables into a read-only rvtable and an execute-only xvtable in order to protect the randomized table contents from disclosure. We modify the Clang C++ front end to split vtables accordingly.

We must also modify all virtual call sites to handle the new, split layout. Rewritten call sites must first dereference the vtable pointer from the object, as usual, to get the address of the correct rvtable. The call must then dereference the xvtable pointer found in this rvtable. After the xvtable address is obtained, the virtual call indexes into this table to find the correct trampoline address for the virtual function, which can then be called. Altogether, we add one additional memory reference to each virtual method call.

In C++, one can also store the address of a class method into a method pointer and later dereference this pointer to call the method. We also handle this slightly more complex case by storing an index in the method pointer struct, as normal. We then handle the xvtable pointer dereference whenever a method pointer is invoked.

Collecting TRaP Information We use several types of information, available at compile time, to properly randomize the PLT and vtable at load time. We embed this meta-data in a special section of the output object files to avoid the need for static binary analysis.

To randomize the xvtables, we need the location of each class's vtable, as well as the number of virtual functions in the class, which is not present in the binary. We modify both the C++ front end and the code emission back end of LLVM to add additional metadata (TRaP information) which marks the location of and number of functions in each vtable for use at load time. We also mark the class inheritance hierarchy for each component of the vtable, since this information is difficult to derive from the vtable alone.

Additionally, we need the location of each reference to the PLT and each virtual call in the program code, in order to rewrite these uses after randomization. We modify the Clang C++ front end to find all PLT references and virtual function call sites and modify the compiler back end to mark these locations in the binary. Specifically, we mark instructions which hold the index of a virtual function so that these indices can be rewritten.

6.2 Runtime Randomization Engine

Our runtime randomization component, `RandoLib`, consumes the TRaP information emitted by the compiler, permutes the PLT and all vtables, and finally updates all ref-

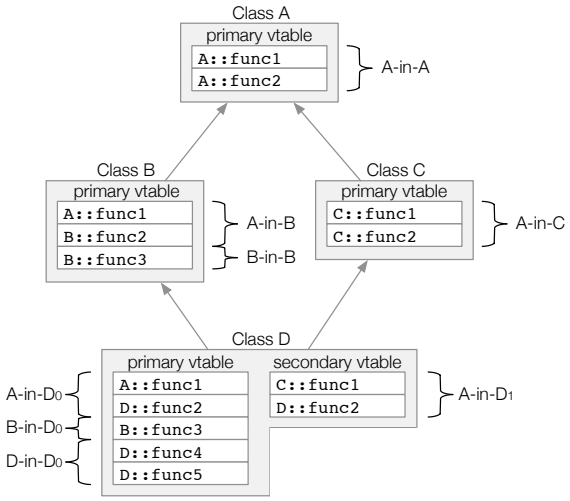

Figure 6: Example class hierarchy rooted in class A. The direct subclasses of A use single inheritance and provide their own implementation of `func2`. Class D uses multiple inheritance and therefore has two virtual function tables.

erences to these tables to refer to a new, randomized index. We built `RandoLib` as a dynamically linked module that randomizes the other loaded modules as the program starts. Our modified compiler automatically emits an initialization routine into each module that registers the module with `RandoLib`. `RandoLib` then randomizes all registered modules together after all modules have been loaded but before the dynamic linker passes control to the program entry point.

`RandoLib` can then be safely unloaded after it has randomized the registered modules. When it unloads, it frees all of the memory it had allocated and overwrites all of the code pointers in its private data with zeros. It therefore does not increase the attack surface.

Identifying tables First, `RandoLib` collects the locations and sizes of the PLT and xvtables from the TRaP section. After identifying all xvtables, we must identify and distinguish all copies of each class's xvtable. Vtables contain sub-vtables corresponding to the vtables of their base classes, and each of these corresponding sub-vtables must be randomized in the same order. To see why this process is necessary, we give an example of a simple inheritance hierarchy and the corresponding vtable layouts. Figure 6 shows the "diamond" inheritance pattern in which a class D has two base classes, B and C that both inherit from a single class A. If a call is made to `func1` on an instance of D, the target of the call depends on the call site. If the call site expects an instance of C, `C::func1` is invoked, and `A::func1` is invoked in all other cases. This behavior is implemented using a primary and secondary vtable for instances of class D (see bottom third of Figure 6). The primary vtable is used from call sites that expect objects of type A, B, or D and the second vtable is used when a call expects an instance of class C. For this reason, all instances of class D contain two vtable pointers.

To explain why xvtables must be split into sub-xvtables for randomization, let us consider randomizing the class hierarchy shown in Figure 6. Three classes derive directly or indirectly from class A. Each of these subclasses therefore has a set of vtable entries that correspond to the virtual methods defined by class A; we call such sets sub-vtables. The layout of a sub-vtable must be the same for all classes related by inheritance.

In Figure 6, the first sub-vtables shared among the class A and its descendants share the A-in- prefix in their names. A virtual method call that operates on an object of type A can effectively use any entry in the A-in- sub-vtables because any object of one of A's sub-types can be assigned to a pointer of type A. This is why we must ensure that the layouts of each of these sub-vtables are mutually compatible.

Since the TRaP section contains the locations and sizes of whole xvtables, `RandoLib` must first split the xvtables into sub-xvtables and then group sub-xvtables together by their type. `RandoLib` relies on the vtable inheritance relationships to identify sub-xvtables. Consider again the previous example. If we know that B's xvtable is based on A's xvtable, then the compiler will guarantee that the lower part of B's xvtable will be the A-in-B sub-xvtable.

Information about the inheritance relationships between xvtables is not readily available at run time. This information can in principle be derived from the Run-Time Type Information (RTTI) structures that are embedded in C++ programs by default [9]. During the course of this research, however, we have found several ambiguous cases where unused vtables are not emitted and therefore we could not always correctly disambiguate complex vtable layouts. Additionally, we found that compilers implementing the Itanium ABI do not always agree on sub-vtable layouts, although this is presumably due to compiler bugs. We have therefore chosen to embed the names of the bases for each sub-vtable in the TRaP information as well. For example, we mark classes A, B, D as the bases of the primary vtable of Class D in Figure 6. This information, combined with the RTTI structures, is sufficient to recover the whole xvtable hierarchy in the program. If it is undesirable to embed C++ RTTI information in the binary, we can easily extend the TRaP information to store all necessary meta-data.

Identifying references Next, `RandoLib` resolves the target of all table references. These references must be updated to reflect the new table layouts after permutation.

The TRaP section contains the locations of all index operations into the PLT and vtables. To save space we do not embed the indices used in these references in the TRaP section, since they can be read by disassembling the marked instructions. We can disassemble references quickly and accurately because they are valid, single instructions.

The TRaP section also contains the name of the intended target class for each xvtable reference. `RandoLib` can then resolve the target to the set of sub-xvtables corresponding to receiving class, which are all randomized using the same permutation as described previously.

Randomization Finally `RandoLib` randomly permutes all tables identified in the first step and updates the references identified in the second step.

We use Algorithm 1 to update xvtable references. In this algorithm, G is a directed graph that represents the class hierarchy. We calculate G based on the RTTI information. Each node in G represents a class that has an array of sub-xvtable pointers ($VTables$). For each sub-xvtable, we store a link to the sub-xvtable that preceded it in the original (whole) xvtable ($Preceding$) and the offset of the sub-xvtable within that original xvtable ($Offset$). $XRefsVector$ is an array of

($Location$, $ClassName$, $Index$) tuples. Each tuple represents one xvtable reference. The $ClassName$, which represents the name of the intended object type for the virtual method call, as well as the $Location$ are read from the TRaP section. We determine the $Index$ by disassembling the reference.

For each xvtable reference, we start by finding the primary xvtable for the call's intended target type (line 4). We then resolve the reference to an exact sub-xvtable group based on the sub-xvtable offsets (line 5-7). We then calculate the index of the intended target of the reference within that sub-xvtable and update the reference (line 8-10).

Algorithm 1 Update vtable references after randomization.

```
1: function AdaptVirtualCalls(G, VCallVector)
2:     for all XRef in XRefsVector do
3:         Class ← G.Find(XRef.ClassName)
4:         SubVtbl ← Class.VTables[0]
5:         while XRef.Index < SubVtbl.Offset do
6:             SubVtbl ← SubVtbl.Preceding
7:         end while
8:         Idx ← XRef.Index − SubVtbl.Offset
9:         Tmp ← SubVtbl.Permutation[Idx]
10:        XRef.Index ← SubVtbl.Offset + Tmp
11:    end for
12: end function
```

7. SECURITY EVALUATION

We aim to prevent attacks utilizing whole-function reuse such as RILC and COOP; our techniques complement the existing Readactor approach which was shown by Crane et al. to be resilient to all known ROP variants in realistic attack scenarios [11]. Here, we evaluate the defensive strength of Readactor++ in the face of an adversary aiming at mounting a whole-function reuse under the conditions described in Section 4.

Readactor provides the invariant that, modulo side channels and plain guessing, the adversary cannot disclose any code pointers except for those pointing to trampolines. However, given control over an indirect branch, knowledge of the address of a function's trampoline is still sufficient for the adversary to invoke that function. Locating a function's trampoline is in many cases simple for the adversary, because trampoline pointers are typically stored at fixed offsets in vtables and other readable data structures such as the PLT. Accordingly, this invariant does not conceptually hinder whole-function reuse.

With Readactor++ in place, the adversary can still disclose the addresses of trampolines that get stored into readable structures. However, as Readactor++ randomizes the layouts of vtables and the PLT, identifying the matching trampoline to a function becomes a challenge for the adversary. Neglecting possible side channel attacks, we observe that an adversary can follow two strategies to overcome this obstacle: (i) reusing functions whose trampoline pointers are stored in large non-randomized and readable data structures (e.g., an application may employ custom function pointer tables with fixed or predictable entries); or (ii) guessing of entries in randomized xvtables or the PLT. Readable tables of trampoline pointers are problematic, and we propose to extend the Readactor++ protections to these through either source code modification, or detection and automatic rewriting during compilation, when possible. We discuss this case further in Section 9. We examine the probability of the latter strategy, guessing of randomized tables' entries, next.

7.1 Guessing Table Layouts

Our adversary model (Section 4) assumes a *brute-forcing mitigation* that permanently terminates an application after a booby trap was hit. Booby traps will not be hit during correct program execution. Even benign programming errors have extremely low likelihood of accidently triggering a booby trap, since we place booby traps in tables the programmer should never access directly.

Since hitting a booby trap will terminate the attack, a successful adversary needs to make an uninterrupted sequence of *good guesses*. What exactly constitutes a good guess depends on the concrete attack scenario. In the best case, the adversary always needs to guess a particular entry in a particular xvtable or the PLT; in the worst case, a good guess for the adversary may be any entry that is not a booby trap. Considering the nature of existing COOP and RILC attacks [31, 39], we believe that the former case is the most realistic. Further, assuming in favor of the adversary that he will only attempt to guess entries in tables with exactly 16 entries (the minimum), we can roughly approximate the probability for Readactor++ to prevent an attack that reuses n functions with $P \approx 1 - (\frac{1}{16})^n$. Our experiments in the following indicate that an attacker needs at least two or three hand-picked functions (from likely distinct tables) to mount a successful RILC or COOP attack respectively. Thus, the probability of preventing these attacks is lower bounded by $P_{RILC,min} \approx 1 - (\frac{1}{16})^2 = 0.9960$ and $P_{COOP,min} \approx 1 - (\frac{1}{16})^3 = 0.9997$.

7.2 Practical Attacks

To evaluate the practical strength of Readactor++, we reintroduced an exploitable bug (CVE-2014-3176) to a recent version of Chromium on Ubuntu 14.04 64-bit. The vulnerability allows an attacker to manipulate JavaScript objects in memory and, consequently, to construct an arbitrary memory read-write primitive. This can be used to reliably disclose memory pointers and hijack the control flow.

We created RILC and COOP exploits for the vulnerability. As is common practice, our exploits change the protection of a memory region in order to execute injected shellcode. For a successful RILC attack, an attacker must correctly guess two function pointers. The first function loads all needed arguments into the register and the second is the memory protection function. In contrast to RILC attacks where the gadgets are chained implicitly through return instructions, COOP attacks require an extra function to chain the gadgets. This third function may be a conventional ML-G, a REC-G, or an UNR-G (see Section 3). In our experiments we successfully executed both attacks, including all COOP variants, on an unprotected version of Chromium. After we applied Readactor++ all attacks failed, as expected.

8. PERFORMANCE EVALUATION

We evaluate the performance of Readactor++ on computationally-intensive code with virtual method dispatch using the C++ benchmarks in SPEC CPU2006. Additionally we test Readactor++ on a complex, real-world C++ application, the Chromium web browser. Overall, we find that Readactor++ introduces a minor overhead of 1.1%. We measured this slowdown independently of the slowdown introduced by the

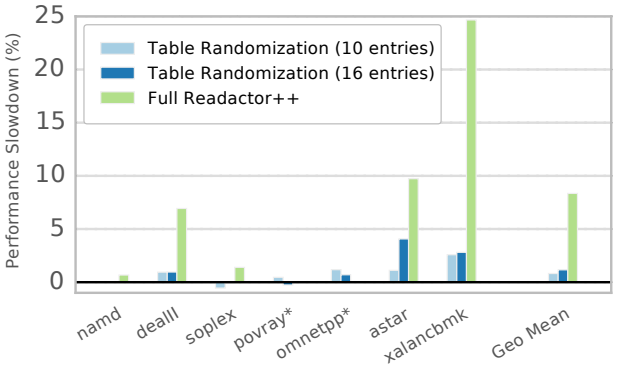

Figure 7: Overhead of table randomization on SPEC CPU2006 C++ benchmarks.

execute-only system itself, which depends on the protection system used and whether hardware natively supports execute-only memory. For a complete system evaluation we also used the Readactor system to enforce execute-only memory and hide code pointers [11]. However, even with this additional slowdown, we find that Readactor++ is competitive with alternative mitigations with an average overhead of 8.4% on SPEC, while offering increased security.

In particular, Readactor++ compares favorably to the performance of recent CFI implementations, after accounting for the need to protect return addresses from disclosure and modification using a shadow stack. VTV [38], a C++ aware forward-CFI implementation, which can thus defend against COOP, incurs an average geometric mean overhead of 4.0% on the SPEC CPU2006 C++ benchmarks using comparable optimization techniques. Dang et al. [12] report that a protected traditional shadow stack, necessary to defend against an attacker with arbitrary memory read/write capabilities, incurs an average overhead of 9.7% on SPEC CPU2006. Thus, the comparable overhead to fully protect against both traditional ROP attacks and COOP attacks using state-of-the-art CFI is 13.7%, in contrast to our average total overhead of 8.4%.

We performed our performance evaluation on an Intel Xeon E5-2660 server CPU running at 2.20 GHz with Turbo Boost disabled running Ubuntu 14.04. To properly evaluate the impact of our transformations on the entire program as well as all relevant libraries, we applied our protections to libc++ and used this library as the standard C++ library. The baseline uses an unprotected version of libc++ to avoid any differences due to variations between C++ standard library implementations.

SPEC CPU2006.

Figure 7 shows the results of our evaluation on SPEC CPU2006. We first evaluated our transformations independently of any execute-only system, shown in the first two columns. For a smaller minimum xvtable size of ten entries ($n_{min} = 10$), we observed a geometric mean slowdown of 0.8%. We observed a 1.1% geometric mean performance slowdown with a minimum xvtable size of $n_{min} = 16$. Since the additional performance slowdown is so minimal, we recommend a minimum xvtable size of at least 16 entries.

We also evaluated Readactor++ in combination with Readactor's execute-only memory protection and code-pointer hiding (labeled Full Readactor++ in Figure 7). For this test we used a minimum vtable size of 16 entries. We observed an average performance slowdown of 8.4% for the combined system on the C++ benchmarks. Crane et al. report a performance overhead of 6.4% for their full system on all of SPEC CPU2006, which is comparable to our results.

Chromium.

To show the impact of our transformations on a complex, real-world application, we built and protected the Chromium web browser.[1] Since our defenses target the C++ components of the browser and not the JavaScript engine in particular, benchmarking the JavaScript performance of Chromium, as most browser benchmarks do, is not useful. Instead, we use the Chromium internal performance test suite to measure scrolling smoothness. This measures the overall performance of the browser when rendering web content. The test suite consists of 25 popular websites, chosen by the Chromium developers, including 13 of the Alexa USA Top 25 sites; Google properties such as Google search, GMail, and Youtube; news websites such as CNN and Yahoo; and Facebook. To account for system noise, we ran each benchmark three times and calculated the geometric mean over these runs. We used a minimum xvtable size of at least 16 entries for this experiment.

We found that the table randomization component of Readactor++ incurred a geometric mean overhead of 1.0%. This measurement was independent of any execute-only memory protection. To evaluate the full impact of Readactor++ with execute-only memory and code-pointer hiding, we combined table randomization with the full Readactor system, including small modifications to disable stack unwinding which is incompatible with Readactor code-pointer hiding. With all Readactor and Readactor++ protections enabled, we measured a combined geometric mean overhead of 7.9%.

9. DISCUSSION

Extensions: We focus on protecting the two main targets of function-reuse attacks: dynamic linking tables and C++ vtables. However, similar tables sometimes exist in other contexts where dynamic dispatch is required. For example, C programs which emulate a variant of object orientation sometimes keep tables of function pointers to perform virtual dispatch, depending on the type in question. Previous work has explored randomizing the layout of data structures [23], and these techniques could be extended to randomize structures or arrays of function pointers. In combination with code-pointer hiding, data structure randomization could protect these pointers from disclosure and reuse.

Dynamic loading of libraries via `dlopen` on Linux and analogous methods on Windows is a special case of dynamic linking. The `dlopen` function is used to load a new library at run time, after the program has started. The program can then use the `dlsym` function to retrieve the address of an exported function from the newly imported library. We can extend Readactor++ to randomize any C++ libraries imported during execution and update all relevant unmodified call sites referring to classes from the imported library.

Limitations: We support all C++ programs that comply with the language specification and do not rely on compiler-specific vtable implementation details. Rare C++ programs which parse or modify their own vtables would need minor modification in order to handle our new split vtable layout.

[1]Chromium sources checked out on 2014-11-04.

In practice we have not seen this issue, since vtables are not a part of the C++ standard at all and vary between compilers, e.g. Itanium-style vtables on Linux and MSVC vftables on Windows. Thus, compiler-agnostic programs must not rely on vtable structure.

Due to these application binary interface (ABI) incompatibilities, programs which import C++ library interfaces must always be compiled with the same C++ ABI version as the external library. Since we modify the vtable portion of the ABI to split vtables, we must also recompile any C++ dependencies with the same ABI. In practice, the only external dependency we found for Chromium or SPEC was the C++ standard library. We rebuilt the libc++ standard library with our modified compiler without any difficulty.

10. RELATED WORK

The literature on memory corruption and countermeasures is vast. One line of defense aims at preventing corruption of code pointers [21, 25]. Most other defenses stop later stages of an attack by enforcing control-flow integrity properties or randomizing the code layout to prevent code reuse. We relate our work to previous defenses of each kind and focus on those most similar to Readactor++.

Code Layout Randomization.

Address space layout randomization (ASLR), a weak form of diversity [32, 33], is widely deployed today and many diversification approaches with finer granularity have been proposed in the literature (see Larsen et al. [22] for an overview). Specifically relevant to this work is the previous proposal to randomize the order of elements in the PLT [4]. Unfortunately, without execute-only code memory most approaches are vulnerable to just-in-time code reuse [35] and other attacks that rely on information disclosure [6, 34].

To defeat JIT-ROP attacks, Backes and Nürnberger [2] propose Oxymoron which hides direct code references embedded in direct call and jump instructions. Hence, an attacker can no longer follow these references to identify and disassemble valid code pages. Unfortunately, Oxymoron can be bypassed with an improved JIT-ROP attack that exploits indirect memory disclosure [13]. The improved JIT-ROP attack is possible because code pages still remain readable in Oxymoron. As a consequence, several schemes have been recently presented that are based on marking code pages as non-readable such as XnR (eXecute-no-Read) [3] and HideM [17]. However, both schemes suffer from limitations: HideM requires a split-TLB based architecture which no longer exists in modern processor architectures. As pointed out by Crane et al. [11], XnR leaves some code pages readable, and requires very fine-grained randomization to mitigate indirect memory disclosure. To tackle these shortcomings, Crane et al. [11] present Readactor which hides code pointers through a layer of indirection and uses Intel's Extended Page Tables (EPT) feature to enable hardware-enforced execute-only memory rather than emulating the feature. However, the readable trampoline pointers still provide a code base which an attacker can exploit to perform a classic RILC or advanced COOP attack [31]. We thwart these function-reuse attacks by permuting all function tables such as C++ virtual tables and PLT entries with performance overheads that allow industry adoption.

Davi et al. [13] recently presented an alternative defensive technique, Isomeron, which combines fine-grained ASLR with control-flow randomization. In particular, it maintains two copies of a program image of which one is diversified. The program control flow is randomized at each function call. However, Isomeron adds significant performance overhead since it leverages dynamic binary instrumentation. In addition, it provides limited protection against RILC and COOP attacks because its policies restrict indirect calls to target a valid function the program links to.

Finally, Mohan et al. [24] present Opaque CFI (O-CFI), a binary instrumentation-based solution which combines coarse-grained CFI with code randomization. Similar to Isomeron, the code layout is no longer a secret. O-CFI works by identifying a unique set of possible target addresses for each indirect branch instruction. Afterwards, it leverages the per-indirect branch set to restrict the target address of the indirect branch to only its minimal and maximal members. To further reduce the set of possible addresses, it arranges basic blocks belonging to an indirect branch set into clusters (so that they are located nearby to each other), and also randomizes their location. Unlike Readactor++, the security of O-CFI relies on the precision of the available CFG. Mohan et al. use CFGs recovered from binaries which leads to coarse-grained policies that may allow function-reuse attacks.

Control-Flow Integrity.

To defend against vtable-based attacks, a number of defenses have recently been proposed [16, 20, 27, 38, 40]. The compiler-based approaches of Tice et al. [38] and Jang et al. [20] focus on protecting virtual function calls in C++. Both ensure that an adversary cannot manipulate a vtable pointer so that it points to an adversary-controlled, fake vtable. Unfortunately, these schemes do not protect against classical return-oriented programming attacks which reuse return instructions. The aforementioned approaches require the source code of the application which might not be always readily available. In order to protect binary code, a number of forward-edge CFI schemes have been presented recently [16, 27, 40]. Although these approaches require no access to source code, they are not as fine-grained as their compiler-based counterparts. In particular, COOP undermines the assumptions of these binary instrumentation-based defenses by invoking a chain of virtual functions through legitimate call sites to induce malicious program behavior [31].

Code-Pointer Integrity.

Recently, Szekeres et al. [37] proposed code pointer integrity (CPI), and Kuznetsov et al. [21] evaluated several implementations for x86 and x86-64 systems. CPI separates code pointers as well as pointers to code pointers from non-control data by placing them in a safe memory region that can only be accessed by instructions that are proven to be safe at compile-time. CPI operates very efficiently on C code, but may incur performance overheads of more than 40% for C++ applications. With respect to security, CPI relies on the protection of the safe memory region which is efficiently possible on x86 by leveraging segmentation. However, on x86-64 where segmentation is not fully available, CPI protects the safe region though information hiding or software-fault isolation. Evans et al. [15] recently demonstrated a weakness in one of the x86-64 CPI implementations that can be leveraged to locate and compromise the safe region.

11. CONCLUSIONS

While ROP-based attacks have received considerable attention from the research community, sophisticated attacks such as COOP show that whole-function reuse is equally worrisome. Our paper demonstrates two new ways to construct COOP attacks against C++ code and describes a minimized yet realistic COOP attack that bypasses DEP. We also introduce a novel probabilistic defense against COOP and other attacks that abuse dynamically-bound function calls. Our techniques are designed to fully resist information disclosure vulnerabilities. Our evaluation shows that these techniques provide quantifiable and tunable protection, scale to real-world software, and add an average run-time overhead of just 1.1%. When combined with execute-only memory and fine-grained code randomization, the combined overhead (8.4%) is less than that of a fully comparable CFI solution.

Acknowledgments

We acknowledge Sajo Sunder George's efforts to implement PLT randomization and boobytrapping and thank Andrei Homescu, Stefan Brunthaler, and the anonymous reviewers for their suggestions and constructive feedback.

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.

This work has been co-funded by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302 (EUREKA project SASER), 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 (project PRACTICE), and the Intel Collaborative Research Institute for Secure Computing at TU Darmstadt.

Finally we thank the Agency for Innovation by Science and Technology in Flanders (IWT) for their support.

References

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

[2] M. Backes and S. Nürnberger. Oxymoron - making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[3] 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 Conference on Computer and Communications Security (CCS)*, 2014.

[4] S. Bhatkar and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

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

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

[7] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

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

[9] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ Application Binary Interface (ABI), 2001.

[10] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *Workshop on New Security Paradigms (NSPW)*, 2013.

[11] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

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

[13] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[14] 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 *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[15] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[16] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

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

[18] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[19] 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.

[20] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[21] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Security Symposium*, 2014.

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

[23] Z. Lin, R. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.

[24] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[25] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[26] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[27] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[28] rix. Smashing C++ VPTRS. *Phrack Magazine*, 56(8), 2000. URL `http://phrack.org/issues/56/8.html`.

[29] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15, 2012.

[30] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[31] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[32] F. J. Serna. The info leak era on software exploitation. In *BlackHat USA*, 2012.

[33] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[34] J. Siebert, H. Okhravi, and E. Söderström. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[35] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[36] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and protecting dynamic code generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[38] 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 *USENIX Security Symposium*, 2014.

[39] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[40] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

APPENDIX

Figures 8–11 below depict excerpts from widely used C++ software resulting in REC-Gs. Each REC-G exists in at least the x86-64 version of the respective software. In each figure, the call sites Ⓐ and Ⓑ are marked analogously to Figure 1.

```
virtual name *clone() const
{
    plural_ptr op1_copy(op1->clone());  Ⓐ
    plural_ptr op2_copy(op2->clone());  Ⓑ
    return new name(op1_copy, op2_copy);
}
```

Figure 8: `mo_lambda.cpp` in the Boost library (version 1.58 and others); causes a side-effect free REC-G on Ubuntu 14.04 and Windows 10.

```
QBlittablePlatformPixmap::
    ~QBlittablePlatformPixmap()
{
#ifdef QT_BLITTER_RASTEROVERLAY
    delete m_rasterOverlay;  Ⓐ
    delete m_unmergedCopy;   Ⓑ
#endif
}
```

Figure 9: `qpixmap_blitter.cpp` in the Qt library (version 5.5 and others); causes a side-effect free REC-G in `libQt5Gui.so` on Ubuntu 14.04 and a REC-G with side effects in `Qt5Gui.dll` on Windows 10.

```
~_Order_node_base()
{
    delete _M_pReceiveMessage;  Ⓐ
    delete _M_pSendMessage;     Ⓑ
}
```

Figure 10: `agents.h` in Microsoft Visual C++ 2013; causes a side-effect free REC-G in Microsoft's C++ runtime library `msvcp120.dll`.

```
size_t SkComposeShader::contextSize() const
{
    return sizeof(ComposeShaderContext)
        + fShaderA->contextSize()  Ⓐ
        + fShaderB->contextSize(); Ⓑ
}
```

Figure 11: `SkComposeShader.cpp` in Chromium (version 44 and others); causes a side-effect free REC-G in Chromium.