The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later

Victor van der Veen
Vrije Universiteit Amsterdam
vvdveen@cs.vu.nl

Dennis Andriesse
Vrije Universiteit Amsterdam
d.a.andriesse@vu.nl

Manolis Stamatogiannakis
Vrije Universiteit Amsterdam
manolis.stamatogiannakis@vu.nl

Xi Chen
Vrije Universiteit Amsterdam;
Microsoft
xcihen@microsoft.com

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

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

ABSTRACT

In 2007, Shacham published a seminal paper on Return-Oriented Programming (ROP), the first systematic formulation of code reuse. The paper has been highly influential, profoundly shaping the way we still think about code reuse today: an attacker analyzes the "*geometry*" of victim binary code to locate gadgets and chains these to craft an exploit. This model has spurred much research, with a rapid progression of increasingly sophisticated code reuse attacks and defenses over time. After ten years, the common perception is that state-of-the-art code reuse defenses are effective in significantly raising the bar and making attacks exceedingly hard.

In this paper, we challenge this perception and show that an attacker going beyond "*geometry*" (static analysis) and considering the "*dynamics*" (dynamic analysis) of a victim program can easily find function call gadgets even in the presence of state-of-the-art code-reuse defenses. To support our claims, we present Newton, a run-time gadget-discovery framework based on constraint-driven dynamic taint analysis. Newton can model a broad range of defenses by mapping their properties into simple, stackable, reusable constraints, and automatically generate gadgets that comply with these constraints. Using Newton, we systematically map and compare state-of-the-art defenses, demonstrating that even simple interactions with popular server programs are adequate for finding gadgets for all state-of-the-art code-reuse defenses. We conclude with an **nginx** case study, which shows that a Newton-enabled attacker can craft attacks which comply with the restrictions of advanced defenses, such as CPI and context-sensitive CFI.

1 INTRODUCTION

Ever since the advent of Return-Oriented Programming (ROP) [62], a substantial amount of research has explored code reuse attacks in depth. Starting from a relatively simple scheme where return instructions served to link together snippets of existing code (*gadgets*), the code reuse concept was quickly generalized to include forward edges such as indirect calls and jumps [9, 59], and even signal handling [10]. Not surprisingly, defenses kept pace with the attack techniques, and a myriad of increasingly advanced attacks [13, 27, 32, 58] was met by equally advanced defenses. Some of these defenses work by constraining control transfers to a specific set of legal flows [1, 66–68], while others complicate attacks by making it difficult to find reusable code snippets [3, 4, 7, 12, 19, 20, 45, 65]. Yet other defenses protect a program by ensuring the integrity of code pointers [43, 46, 47].

In principle, exploitation may still be possible even in the presence of these defenses; for instance, through implementation issues [14, 26]. However, in practice, code-reuse attacks on a system with state-of-the-art defenses are extremely challenging. Such attacks require an attacker to analyse the protected program to find available defense-specific gadgets that can be used to implement the desired malicious payload. Crucially, the literature on code reuse attacks has thus far focused on the threat model introduced in Shacham's original work on ROP [62], which is based on (manual or automatic) *static analysis*. This is an important observation, because modern defenses reduce the set of available gadgets to the point that finding a sufficient set of gadgets for an exploit stretches the abilities of even the most advanced static analysis techniques. In this paper, we introduce a novel approach for constructing code reuse attacks even in the presence of modern defenses. The key insight is that the required analysis effort to construct an attack can be greatly reduced and scale across a broad range of defenses by using *dynamic analysis techniques* instead of only static analysis.

Static flesh on the bone. The original paper introducing Return-Oriented Programming appeared at CCS exactly 10 years ago [62], and demonstrated the first general formulation of a code-reuse attack. With ROP, an attacker would use static analysis to scan the binary for useful snippets of code that ended with a return instruction. Out of these code snippets, known as *gadgets*, the attacker would construct a malicious payload and link them by means of the return instructions at the end of the gadgets. By injecting the appropriate return addresses on the stack of a vulnerable program, an attacker could craft arbitrary functionality.

All code-reuse techniques since have followed this same basic approach—using static analysis to first identify which gadgets are available, and then constructing a malicious payload out of them.

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 ACM 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'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/http://dx.doi.org/10.1145/3133956.3134026

This is true even for advanced exploitation that performs such analysis "just in time" [63].

Modern code-reuse defenses push such attacks to their limits and the analysis required to bypass them is now highly sophisticated [58, 59]. In the absence of implementation errors or side channels, an attacker would be hard-pressed to locate the gadgets, let alone stitch them together. In other words, state-of-the-art defenses have been successful in raising the bar: they may not stop all possible attacks, but they make them exceedingly difficult.

Beyond static analysis. The key assumption for the effectiveness of current defenses is that future attacks follow essentially the same—static analysis-based—approach as proposed by Shacham in 2007. In this paper, we challenge this assumption, demonstrating that a switch of attack tactics to include dynamic analysis renders current defenses far less effective and attacks far less laborious. In reality, attackers do not care about gadgets or ROP chains—all they want is to execute a sensitive call such as `execve` or `mprotect` with arguments they control. There is no reason to assume that they would limit themselves to static analysis.

The goal of modern defenses is to prevent attackers from subverting a program's control flow to reach a desired target, even if an attacker is able to read or write arbitrary data. The question that the attackers must answer is which memory values they should modify to gain control over the program. Ideally, they would answer this question without resorting to complex static analysis.

The key insight in this paper is that we can model such an attacker's capabilities by means of dynamic taint analysis. In particular, we taint all bytes that an attacker can modify with a unique color and then track the flow of taint until we reach code that, given the right values for the tainted bytes, allows the attacker to launch a code-reuse attack. For instance, if a tainted code pointer and a tainted integer later flow into an indirect call target and its argument (respectively), we have concrete evidence that the attacker can fully control a particular call instruction "gadget". Shacham's original static analysis tool is named *Galileo*, a play on its use of "*geometry*". Since our approach is largely based on dynamic ("*dynamics*") rather than static ("*geometry*") analysis, we refer to our gadget-discovery framework as Newton.

As we shall see, our approach requires an attacker to simply run the victim process with Newton's dynamic analysis enabled. Moreover, our approach can easily emulate common constraints imposed by modern defenses against code reuse. Depending on the defense, we may be able to corrupt some locations (but not others) and target some functions (but not others). As detailed later, we can map these per-defense restrictions to simple and stackable constraints (e.g., tainting policies) for our analysis. Moreover, an attacker may model such constraints once and reuse them across a wide variety of defenses and victim applications.

Contributions. Our contributions are the following:

- We show that a hybrid static/dynamic attacker model significantly lowers the bar for mounting code-reuse attacks against state-of-the-art defenses.
- We implement Newton, a novel framework for generating low-effort code-reuse attacks using constraint-driven dynamic taint analysis.
- We evaluate and compare existing defenses against code reuse, highlighting their respective strengths and weaknesses using constraints in Newton.
- We present an `nginx` case study to demonstrate how to use Newton to craft code reuse attacks against advanced defenses, such as secure implementations of CPI [43] and context-sensitive CFI [67].

2 THREAT MODEL

We consider a code-reuse attacker armed with arbitrary memory read and write primitives based on memory corruption vulnerabilities (e.g., CVE-2013-2028 for `nginx` and CVE-2014-0226 for Apache), similarly to recent work [19, 46, 52, 58, 61]. We focus on a low-effort attacker, relying on such primitives and automatic gadget-discovery tools to craft attacks with limited application knowledge. Our attacker seeks to locate gadgets and mount code-reuse attacks, even in the face of state-of-the-art defenses such as Control-Flow Integrity (CFI) [66–68], leakage-resistant code randomization [12, 19], and Code-Pointer Integrity (CPI) [43]. We focus specifically on lightweight code-reuse defenses and leave more general heavyweight defenses such as memory safety [48, 49] or Multi-Variant Execution (MVX) [41, 69] out of scope.

Given the overwhelming number of code-reuse defenses in the literature, we limit our analysis to only (1) defenses applicable to general programs (e.g., no vtable protection for C++ programs [66]), (2) the strongest designs in each class (i.e., effectiveness against weaker defenses is implied), and (3) the secure implementation of such designs (e.g., no side-channel [26, 52] or weak-context [14] bypasses). We also assume a strong baseline with ASLR [55], DEP [2], a perfect shadow stack [21] (making it impossible to divert control-flow by modifying return addresses), and coarse-grained forward-edge CFI [75] (callsites can only target function entry points) enabled.

We assume that the attacker has access to a binary equivalent to the one deployed by his prospective victim. Finally, for simplicity, we focus specifically on popular server programs, similar to much prior work in the area [8, 46, 51, 52, 58, 61, 67, 68].

3 OVERVIEW OF CODE-REUSE DEFENSES

In this section, we provide an overview of state-of-the-art code-reuse defenses considered in our threat model. We distinguish four classes of code-reuse defenses: (1) Control-Flow Integrity, (2) Information Hiding, (3) Re-randomization, and (4) Pointer Integrity. We now introduce each of these classes in turn, and later show how to map them to Newton constraints in §5.

Control-Flow Integrity. The idea of (forward-edge) Control-Flow Integrity (CFI) is to mitigate code-reuse attacks by instrumenting indirect callsites to ensure that only legal targets allowed by the (inter-procedural) Control Flow Graph (CFG) of the program are permitted [1]. To determine the targets for each callsite, modern CFI solutions use either static or dynamic information.

CFI solutions that rely only on static information either allow callsites to target all function entry points [74, 75] or, more recently, construct the set of legal targets by mapping callsite types to target function types. In other words, a callsite of the form `foo(struct bar *p)` should only call functions of type `func(struct bar`

`*p)`. In particular, IFCC [66] and MCFI [50] construct such mappings using source type information, while TypeArmor [68] approximates types based on argument count at the binary level.

CFI solutions that rely on dynamic information track execution state to improve the accuracy of static analysis. In particular, PICFI [51] implements a "*history-based CFI*" (*HCFI*) policy, restricting the target set to function targets whose address has been computed at runtime. Context-sensitive CFI (*CsCFI*) solutions (or similar, with different definitions of "context") such as PathArmor [67], GRIFFIN [28], FlowGuard [44], kBouncer [54], and ROPecker [16] restrict the target set based on analysis of the last n branches recorded by hardware, e.g., the Last Branch Record (LBR) registers or Intel PT. The effectiveness depends on the amount of useful "context" in the branch history, which is necessarily limited in practical implementations: 16 or 32 LBR entries [16, 54, 67], 30 Intel PT packets [44], or a limited policy matrix [28].

Information Hiding. Information hiding (IH) aims to prevent code reuse by making the locations of gadgets unknown to an attacker. This is done by (1) diversifying the code layout using traditional Address-Space Layout Randomization (ASLR) [55] or more fine-grained variants [4–6, 12, 15, 17–20, 29, 31, 35, 36, 40, 42, 53, 64, 71] and (2) "hiding" code pointers to an arbitrary memory read-enabled attacker. The latter property is enforced in different ways by different leakage-resistant randomization solutions.

Oxymoron [4] removes all the code references from the code, preventing an attacker reading any given code page from gathering new code pointers that reveal the location of other code pages. Other solutions such as Readactor [19], software-based XnR [3], HideM [30], LR^2 [12], KHide [29], kRˆX [56], Heisenbyte [65], and NEAR [72] implement eXecute-Only Memory (*XoM*) or similar semantics for code pages, preventing an attacker from reading useful gadgets from the code and thus fully "hiding" the code layout (in the ideal case). Finally, recent solutions such as Readactor++ [20] and CodeArmor [15] extend XoM semantics (*XoM++*) to also hide code pointer tables such as the Global Offset Table (GOT).

Re-randomization. Re-randomization (RR) is another popular defense strategy against code reuse attacks. Unlike information hiding, re-randomization solutions seek to re-randomize and invalidate leaked information (ideally) before the attacker has a chance to use it and craft just-in-time code reuse attacks [63]. Existing solutions can be classified based on the particular information they periodically re-randomize during the execution.

A number of RR solutions such as Shuffler [73], CodeArmor [15], and ReRanz [70] periodically re-randomize the code layout (*CodeRR*) but leave the function pointer values stored in data pages (heap, stack, etc.) immutable using indirection tables. In contrast, TASR [7] re-randomizes each code pointer value in memory every time the corresponding code target is re-randomized. Finally, other solutions such as ASR_3 [31] and RuntimeASLR [45] re-randomize the full memory address space layout, including the values of code and data pointers at each re-randomization period.

Pointer Integrity. Pointer integrity (PI) solutions seek to counter code reuse by preventing attackers from tampering with code or data pointers. Existing solutions can be classified as encryption-based or isolation-based.

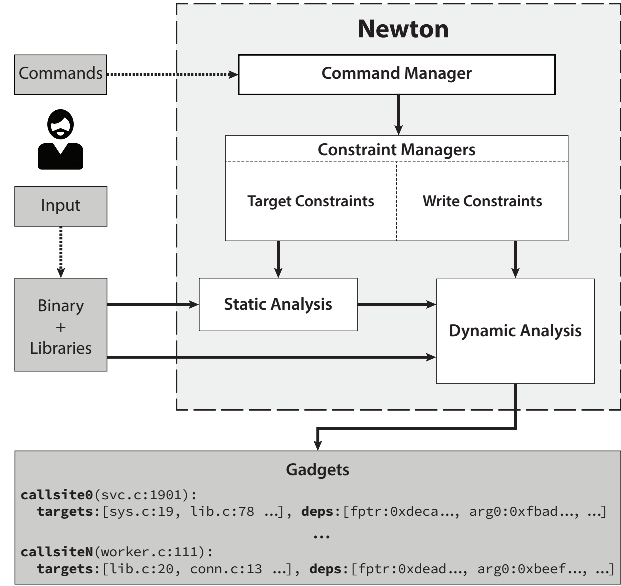

Figure 1: Design of Newton.

ASLRguard [46] is an encryption-based solution that encrypts each computed code pointer with a per-pointer key in a safe vault, (ideally) preventing attackers from crafting new code pointers in memory. In contrast, CCFI [47] encrypts each code pointer stored into a given memory address with an address-dependent key, also preventing attackers from reusing leaked code pointers in memory.

CPS [43] is an isolation-based solution that isolates all the code pointers in a protected safe region, (ideally) preventing an attacker from reaching and corrupting any of these pointers. CPI [43] extends CPS to also isolate data pointers that may indirectly be used by the program to access code pointers, (ideally) preventing an attacker from corrupting code and related data pointers in memory.

4 OVERVIEW OF Newton

We now present Newton, our gadget-discovery framework to assist in crafting code-reuse attacks against arbitrary (modeled) defenses. For this purpose, Newton applies a uniform and black-box strategy to dynamically retrieve gadgets as a set of *attacker-controllable forward CFG edges*. Each edge is expressed as a callsite with a number of possible target functions, and tagged with a number of *dependencies* (e.g., the target function is controlled by the code pointer stored at address X and the first argument is controlled by address Y). These edges can then be inspected by an attacker and used to call arbitrary functions via arbitrary memory read/write primitives. To call a sequence of arbitrary functions, an attacker can chain a number of such edges together over multiple interactions with the victim application.

To easily support a broad range of code-reuse defenses, Newton accepts a number of user-defined constraints that limit the analysis to only gadgets allowed by the given modeled defense. The key idea is to run the victim program mimicking the stages of the real attack and constrain Newton's dynamic gadget analysis using simple, reusable, and extensible policies that map the

security invariants of a broad range of defenses. We discuss the mapping of defenses to constraints later, in §5.

Figure 1 presents an overview of Newton and its high-level components. The Newton framework pushes the victim binary and its shared libraries through a pipeline of (1) static analysis, and (2) dynamic analysis—on top of a *dynamic taint analysis (DTA) engine*. During both phases, the *target* and *write* constraint managers apply user-defined constraints to the analysis, eventually yielding a list of callsites an attacker can control and, for each callsite, a list of callees an attacker can target under a given defense (or combination of defenses) regime.

In more detail, the workflow of Newton when analyzing a binary to craft a code reuse attack is as follows.

(1) At the start of the analysis process, the user starts the target application binary normally. At this point, Newton is in a waiting state, and does not yet perform any analysis.
(2) The user now brings the application into a stable state where they can effect arbitrary memory read/write primitives. In our evaluation, we assume that the user brings the victim program into a simple *quiescent state*. For instance, in the case of a server application, the user would perform a minimal set of interactions to bring the server into an idle state with an open connection, where only long-lived data persists in memory, as in [52]. In general, the chosen quiescent state is program-dependent.
(3) Next, the user signals Newton that the victim application is now in a quiescent state. At this point, Newton begins tracking user-controlled memory dependencies using its DTA engine.
(4) At the same time, the user supplies Newton with a number of commands (in a script) to specify the target and write constraints that Newton should assume are used to defend the victim application. As a result, Newton will take these constraints into account during its analysis of controllable edges.
(5) The user now interacts with the victim application, using the inputs they want to use during the final exploit. This allows Newton to track the dependencies during these interactions. Focusing on a low-effort attacker targeting a server application, we assume that the interactions amount to simple standard requests to the victim server.
(6) Finally, Newton reports the results of its analysis. This yields a set of gadgets (callsites+targets) that are under the user's control given the user's chosen defense model, initial quiescent state and set of server interactions.

4.1 Constraints

As defined by our threat model, our goal as an attacker is to use an arbitrary memory read/write primitive to divert control flow. The baseline defenses described in §2 force us to achieve this by corrupting memory in such a way that later in the execution, the target of an indirect callsite no longer points to its intended callee. With this in mind we observe that, conceptually, all existing defenses attempt to avert successful attacks by enforcing constraints along one (or both) of the following two dimensions:

(1) **Write constraints.** Write constraints limit an attacker's capability to corrupt writable memory. Without any defense deployed, an attacker can corrupt anything: (1) pointers to code (function pointers), (2) pointers to data, and (3) non-pointer values such as integers or strings.
(2) **Target constraints.** Constraints on targets limit the attacker in his selection for possible callees of a controlled callsite. Without any target constraints beyond the baseline, the target set always consists of all functions in the program and library code. We show later how different defenses and their constraints reduce the wiggle room for an attacker.

4.2 Write Constraint Manager

The write constraint manager accepts user-defined constraints, describing the memory regions the attacker is allowed to overwrite under the modeled defense. Then, using *constraint-driven dynamic taint analysis*, it pinpoints callsites and arguments which can still be controlled by the attacker, despite the assumed defenses. Newton's DTA engine is a heavily modified version of `libdft` [39] which supports arbitrary tags per memory location, as well as additional functionality to support the command manager API (see §4.4). The steps of the analysis are as following:

(1) **Initial tainting.** We model attacker-controlled memory by initially marking regions under the attacker's control as tainted. To easily model different defenses, Newton exposes taint limiting commands that allow control over how the initial taint is applied (see §4.4). Newton's DTA engine propagates the taint information to callsites and arguments.
(2) **Tracking dependencies.** We configure our taint engine with a unique tag for each byte in memory, allowing us to track attacker-controlled memory dependencies at byte granularity. Our dynamic taint analysis engine is capable of tracking the taint source address for each tainted value or pointer in memory. For each tainted byte, this tells us exactly by which memory addresses it was tainted. This allows us to track, when a tainted callsite is discovered, where the taint originated for the associated function pointer and each of the arguments. The source of the taint is then a candidate value for the attacker to corrupt, to control the callsite and mount a code-reuse attack.
`libdft`'s original implementation implements a basic tainting strategy [39], able to track only direct attacker-controlled memory dependencies (i.e., callsite X uses code pointer at tainted address Y) and not indirect ones (i.e., callsite X' uses code pointer read via data pointer at tainted address Y'). To support the latter, we implemented pointer tainting for memory reads in `libdft` [39] (i.e., taint every value read via a tainted pointer), allowing us to model an attacker corrupting data pointers and non-pointers to indirectly control code pointers (and arguments) used by tainted callsites.
(3) **Logging.** When an indirect call is executed, Newton logs the relevant taint information for this callsite. Specifically, for each tainted callsite, we emit information detailing the taint dependencies for the callsite's target, and the first six arguments.

4.3 Target Constraint Manager

Like the write constraint manager, the target constraint manager models constraints imposed by code reuse defenses. It uses static and dynamic analysis to extract callsite and callee information, which it then uses to impose the user-defined constraint policy.

Static analysis. We use a static analysis based on DWARF debugging symbols to extract all callsites and potential callees from the target binary and shared libraries, along with associated type information. Newton uses the extracted information (if instructed) to simulate a number of policies for existing defenses, such as type-based CFI [50, 66, 68].

Dynamic analysis. In addition to the aforementioned static analysis, we also use dynamic analysis to scan user-defined ranges of writable memory (such as `.data`, or the heap) for code pointers. We define a *live code page* as a memory page pointed to by a *live code pointer*, i.e., a code pointer stored in live data memory that can be leaked and overwritten. Our dynamic analysis allows us to track live code pointers and code pages. We use this information to model target constraints imposed by defenses such as Readactor [19], which limit an attacker to using "live" gadgets in memory.

The target constraint manager logs the valid targets for each callsite based on the constraints derived by the static and dynamic analysis, as guided by the user-defined script modeling the defense.

4.4 Command Manager

As mentioned, Newton includes write constraint and target constraint managers which model the constraints imposed by a particular defense, based on a user-defined script. To handle the scripting commands, Newton includes a *command manager*. The command manager is a preloaded library that loads along with the analyzed binary, and listens for commands on a Unix domain socket. When a command is received, the command manager dispatches it to the right constraint manager, which handles it as needed.

Newton exposes the following command functions, sufficient to map all of the defenses we evaluate in §6. In §5, we show examples of these commands used in practice to model defenses.

- **`taint-mem`**: This command instructs the taint analysis engine to mark all writable memory as tainted, simulating the arbitrary read/write primitive we assume in our threat model (see §2). It initializes the source taint for each value to its own address. In §5, we show how among other things, we use **`taint-mem`** to taint all memory after bringing a victim server program into a quiescent state.
- **`taint-flip`**: This command untaints all tainted data, and taints all untainted data. We use the ability to flip taint when crafting history-flushing attacks against context-sensitive CFI defenses, as explained further in §5.
- **`taint-prop-toggle`**: This command pauses or resumes the propagation of taint (also implies **`taint-log-toggle`**) by Newton's DTA engine. *Default*: on.
- **`taint-log-toggle`**: Similarly to **`taint-prop-toggle`**, this command pauses or resumes the logging of tainted callsites. This is used to avoid logging uninteresting callsites. Taint propagation continues normally. *Default*: on.
- **`taint-ptr-toggle`**: This command enables or disables pointer tainting on memory reads. *Default*: on.
- **`taint-wash (CPtr|Ptr|AddressRange)`**: This command clears the taint for particular memory locations: locations with code pointers, data pointers, or in a given address range.
- **`constrain-targets`**: This command specifies target constraints to enforce on tainted callsites.
- **`get-gadgets`**: This command retrieves all gadgets collected during the execution.

5 MAPPING DEFENSES IN Newton

As mentioned in §4, for the purpose of finding gadgets for code reuse with Newton, we model the security provided by code-reuse defenses along two axes: (1) write constraints imposed by the defense, and (2) the imposed target constraints. In this section, we map the defenses from §3 according to these constraints. This mapping allows us to easily create scripts that teach Newton about the constraints (security restrictions) imposed when searching for attacker-controllable gadgets (callsites and possible targets).

5.1 Deriving Constraints

Table 1 summarizes the constraints imposed by each defense class. We now discuss each class in detail.

Control-Flow Integrity. We distinguish five subclasses within the CFI class of defenses: (1) TypeArmor, (2) IFCC/MCFI, (3) Safe IFCC/MCFI, (4) HCFI, and (5) CsCFI.

TypeArmor imposes target constraints which enforce that call sites can only target functions with a type matching the call site's type; such types are approximated by statically analyzing the program binary (*Bin types*). Since TypeArmor is the only defense which offers function type-based CFI at the binary level, it has its own dedicated subclass in Table 1.

The IFCC/MCFI subclass provides similar constraints as the TypeArmor subclass, except that function type information is computed at the source rather than at the binary level. This leads to a stronger target constraint (*Src types*) and hence security. This is because source information allows IFCC/MCFI to compute more accurate type information and derive a smaller legal target set.

Safe IFCC/MCFI comprises the same defenses as the IFCC/MCFI subclass, except that in this case the defenses run in a "safe" mode, where type information is less strict for compatibility reasons with real-world programs—discussed in the original IFCC paper [66]. For instance, in this mode, pointer parameters such as **`int*`** or **`void*`** are each considered to be interchangeable with other pointer types. This leads to a weaker target constraint (*Safe src types*) compared to the non-safe variant of this subclass.

In the HCFI (history-based CFI) subclass, the set of valid targets for each call site is determined by the set of code pointers that have been computed during the execution. This is a dynamic target constraint (*Computed*), which can be used in isolation or combined with other static target constraints.

All the CFI subclasses thus far have been modeled using target constraints. Somewhat counterintuitively, we model the CsCFI subclass using *only* write constraints. The reason is that this makes it much easier to write a Newton CsCFI-aware script for a low-effort

Table 1: Mapping of code-reuse defenses to NEWTON constraints. Empty entries for write/target constraints indicate that the defense imposes no write/target constraints, respectively.

Defense			Write constraints	Target constraints	
Class	*Subclass*	*Solutions*	*Details*	*Dynamic*	*Details*
CFI	TypeArmor	[68]			Bin types
	Safe IFCC/MCFI	[50, 66]			Safe src types
	IFCC/MCFI	[50, 66]			Src types
	HCFI	[51]		✓	Computed
	CsCFI	[16, 28, 34, 44, 54, 67]	Segr		
IH	Oxymoron	[4]		✓	Live +page
	XoM	[3, 12, 19, 29, 30, 56, 65, 72]		✓	Live
	XoM++	[15, 20]		✓	Live ¬GOT
RR	CodeRR	[15, 70, 73]		✓	Live
	TASR	[7]	¬CPtr	✓	Live
	PtrRR	[31, 45]	¬Ptr	✓	Live
PI	ASLR-Guard	[46]		✓	Live
	CCFI/CPS	[43, 47]	¬CPtr	✓	Live
	CPI	[43]	¬Ptr	✓	Live

attacker. Formulating CsCFI in terms of target constraints would require us to provide NEWTON with knowledge about the context-sensitive analysis, the branch history size, and the time of validation (e.g., syscall time). Furthermore, when assuming a "perfect" (but practical) implementation of CsCFI, the branch history can be arbitrarily large (but not unlimited), allowing a "perfect" context-sensitive analysis to always detect invalid targets in the large context provided. In other words, the only way for an attacker to bypass the defense is to force the application to flush the (arbitrarily large) branch history [14] before triggering the exploit. This leaves CsCFI with no context to constrain the controlled target set.

For this purpose, the attacker needs to (1) corrupt some *segregated* (independent and stable) application state, (2) send an arbitrarly large number of idempotent *history-flushing inputs* to the application that do not interfere with the segregated state, (3) send the final input to trigger the exploit based on the segregated state. This translates to a write constraint (*Segr*) that limits writes to the segregated state specified by the attacker. At first glance, identifying such state and the history-flushing input seems complicated. In practice, this is possible even for a low-effort attacker. For example, for common server applications that handle multiple connections in a single worker process (e.g., `nginx`), we can simply instruct NEWTON to use the connection-specific data of a first connection as segregated state and a simple request over a second connection as the history-flushing input (as done in §5.2).

Information Hiding. We distinguish three subclasses within the IH class of defenses: (1) Oxymoron, (2) XoM, and (3) XoM++.

The Oxymoron subclass allows only targets contained in *live code pages*. This translates to a target constraint (*Live +page*) that limits the set of valid (i.e., attacker-leaked) targets to pages pointed to by live code pointers.

The XoM subclass contains defenses that hide the code layout from an attacker. This translates to a target constraint (*Live*) that limits the set of valid targets to live code pointers (again stored and then leaked from memory), given that the attacker can make no assumptions on the other code pointers.

Finally, defenses in the XoM++ subclass implement XoM semantics and additionally hide the GOT from an attacker. This translates to a stronger target constraint (*Live ¬GOT*) than XoM's, where live code pointers in the GOT are no longer valid. Since the GOT itself is no longer reachable and thus not corruptable, this also translates to a write constraint (*¬GOT*), which, for simplicity, we leave implicit in our analysis and presentation of the results (its impact typically aligns with its target constraint counterpart).

Re-randomization. We distinguish three subclasses within the RR class of defenses: (1) CodeRR, (2) TASR, and (3) PtrRR. Since all these subclasses hide the code layout under ideal conditions, they all impose a target constraint that allows only live code pointers to be used as valid targets (*Live*). However, the subclasses differ in terms of their write constraints.

First, the CodeRR subclass only hides (i.e., re-randomizes) the code layout and imposes no additional write constraints. The second RR subclass, TASR, does impose an additional write constraint. Not only does TASR periodically re-randomize the code layout, but it also re-randomizes the code pointer representation (even for code pointers stored in data memory). In doing so, it prevents attackers from successfully overwriting code pointers. This translates to a write constraint (*¬CPtr*) that forbids writes to memory locations containing code pointers. In other words, this constraint teaches NEWTON that the only way to find gadgets that bypass CodeRR is to corrupt data pointers (or non-pointers) to force the program to access an attacker-controlled live code pointer rather than the original intended target (e.g., corrupting `c` to hijack `c->handler()`).

Finally, the PtrRR subclass is similar to TASR, except that the imposed write constraint is stronger. Not only code pointers but *all* pointers are re-randomized and thus "protected" against overwrites. This translates to a write constraint (*¬Ptr*) that forbids writes to memory locations containing either code or data pointers.

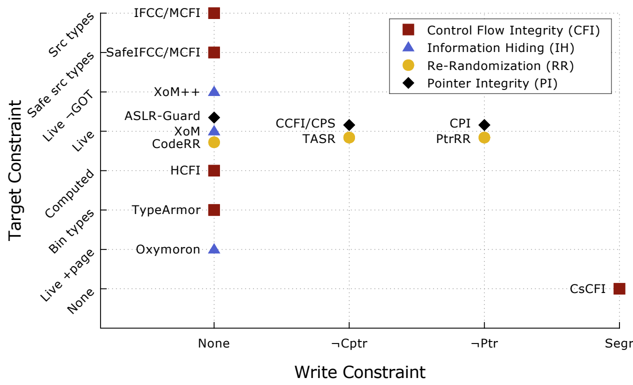

Figure 2: Mapping of defense classes to write (x-axis) and target (y-axis) constraints in Newton. Constraints on the two axes are sorted based on their effectiveness in reducing the number of gadgets available to a low-effort attacker on `nginx`, when sending a plain `HTTP GET` request.

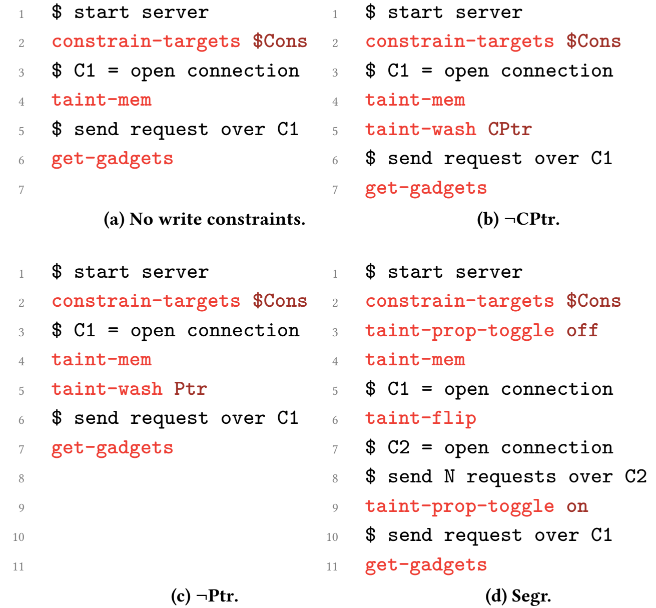

Figure 3: Newton command scripts for finding gadgets under different modeled write constraints.

In other words, this constraint teaches Newton that the only way to find gadgets that bypass PtrRR is to corrupt non-pointers such as integers (e.g., corrupting `idx` to hijack `func[idx]->handler()`).

Pointer Integrity. We distinguish three subclasses within the PI class of defenses: (1) ASLR-Guard, (2) CCFI/CPS, and (3) CPI. All three of these prevent an attacker from crafting new code pointers from scratch, thus enforcing a target constraint that limits targets to live code pointers (*Live*).

ASLR-Guard does not impose any additional constraints. It implements the aforementioned target constraint by using per-pointer secret keys to encrypt all code pointers. Thus, while an attacker cannot introduce new code pointers, it is still possible to replace a code pointer with another arbitrary live code pointer, given that the secret key is not location-aware.

The second PI subclass, CCFI/CPS, does impose an additional write constraint that forbids writes to memory locations containing code pointers (*¬CPtr*). In the case of CCFI (Cryptographically-enhanced CFI), this is implemented by encrypting pointers with a memory location-dependent key. In the case of CPS, the same effect is achieved by isolating code pointers in a memory region not accessible to an attacker.

Finally, CPI is equivalent to CPS, except that it isolates not only code pointers, but also data pointers that point to structures containing code pointers. Thus, CPI imposes a stronger write constraint than CPS, forbidding writes to memory locations containing either code or data pointers (*¬Ptr*).

5.2 Implementation

Figure 2 graphically depicts the constraints imposed by the defenses, as detailed in Table 1. The x-axis shows the write constraints imposed by each defense subclass, while the y-axis shows the target constraints. Defenses that share both the same write and target constraints impose equivalent security restrictions, so that each (x, y) point in Figure 2 forms an *equivalence class.*

It is interesting to note that even defenses that seem quite different on the surface actually turn out to offer comparable guarantees. For instance, the figure reveals the following equivalence classes containing multiple defenses each: $\{XoM, CodeRR, ASLR-Guard\}$, $\{TASR, CCFI/CPS\}$, and $\{PtrRR, CPI\}$. Note that these equivalences hold only when assuming "perfect" implementations of each defense, without any implementation-specific vulnerabilities. In addition, our constraint-based classification abstracts away implementation details and hence ignores implementation-specific differences across defenses. For instance, the ¬Ptr constraint in RuntimeASLR protects all data pointers, and could thus be considered stronger than the same constraint in CPI, which protects only data pointers that can be used to read code pointers. The key advantage of our approach is that it allows us to focus on the general constraints for gadget generation across many different defenses.

We now demonstrate how to concretely implement constraints for the mapped defenses in Newton, using the commands detailed in §4. We organize the following discussion around the write constraints imposed by each defense.

Corrupting code pointers. All defense subclasses that do not implement write constraints allow any memory to be corrupted, including code pointers. These defenses are on the left of the x-axis in Figure 2 (*None*).

To model these defenses, we use the Newton script shown in Figure 3a. All our example scripts assume a low-effort attacker attacking a server application. After starting the server, the script first informs Newton about any target constraints; this guides Newton's static and dynamic analysis of callees and live code pointers. Newton has internal support for each of the possible target constraints shown in Table 1 and Figure 2.

Next, the script taints all memory using the `taint-mem` command. We then send a normal request to the server, causing Newton to track any taint propagated during this request. As the request is processed, Newton logs tainted callsites, their arguments, dependencies, and potential targets. These gadgets can then later be retrieved by the user (`get-gadgets` command).

Corrupting data pointers. Defense subclasses with the ¬CPtr write constraint prevent code pointers from being overwritten, but do not protect other memory locations. This includes the CCFI/CPS and TASR subclasses. As a result, under these defenses, it is still possible to corrupt data pointers (as well as non-pointers).

We model these defenses in Newton using the script shown in Figure 3b. The script is identical to the script we used to model defenses without any write constraints, except that after tainting all memory, we use the `taint-wash` command to untaint code pointers. This has the result of simulating that code pointers are not overwritable by an attacker, thus modeling defenses in the ¬CPtr write constraint class.

Corrupting non-pointers. Under defenses that implement the ¬Ptr write constraint, neither code nor data pointers can be written, limiting the attacker to overwriting only non-pointers. We simulate this using the script shown in Figure 3c, in which we clear the taint for both code and data pointers after tainting memory.

Corrupting segregated state. As mentioned in §5.1, we model the CsCFI subclass using write constraints instead of target constraints, as this makes CsCFI easier to emulate in Newton. As described earlier, the write constraints impose a "segregated memory" defense model, in which an attacker corrupts program state in such a way that this state is not modified by subsequent history-flushing requests. The attacker then uses an arbitrary number of these requests to flush the context of the CsCFI defense, after which it becomes possible to use the previously corrupted state to trigger an exploit.

We model this in Newton using the script shown in Figure 3d. The script begins by starting the victim server and setting the target constraints, as usual. Next, we disable taint propagation, after which we taint all memory and open an attack connection (c_1), and finally flip the taint state of all memory. Opening the connection has the effect of clearing taint on the memory touched by the connection state. Thus, when we flip the taint state, the untainted memory (containing the connection state) becomes tainted, while all other memory becomes untainted. This way, we model the initial segregated (connection) state, which will serve as the attack surface in the final exploit. Note that the segregated state is not an idle state as our attack connection is still open, and that there are possibly many more active open connections in parallel.

We now send an arbitrary number of idempotent requests to the server over an independent history-flushing connection c_2. This is to model flushing the CsCFI context and also ensure there is no interference with the state of connection c_2. Finally, we re-enable taint propagation, resume the attack connection c_1 (left open previously), and send the final request. The final result of the analysis is a list of callsites (with possible targets and dependencies) which are tainted *only* by attacker-controlled connection-specific state, and are thus controllable by the attacker after the history-flushing attack is complete. This voids the concern that some of the long-lived structures in the quiescent state may be modified by parallel connections.

6 EVALUATION

We evaluate Newton against three web servers (`nginx`, Apache, and `lighttpd`), a general-purpose distributed memory cache system (`memcached`), an in-memory database (`redis`), and a domain name system (`bind`). As is common these days, we compile the servers as position independent code, using `gcc` as our compiler.

Using Newton scripts as presented in §5.2, we instruct our target constraint manager to apply each of the target-based policies from §5 (in addition to the baseline as described in §2). As described there, we divide the deployed defenses into those with static target constraints, and dynamic ones.

Also recall from §5.2 that our scripts instruct the write constraint manager to apply the following types of write constraints: (1) None, this is our baseline where an attacker can corrupt anything, including code pointers; (2) ¬CPtr, policies that restrict the corruption of code pointers; (3) ¬Ptr, policies that enforce pointer integrity; and (4) Segr, for context-sensitive CFI.

We first perform a detailed evaluation for `nginx`, in which we provide statistics on the controllability of each executed indirect callsite. Later, in §7, we show how to use this information to mount defense-aware attacks against `nginx`. In the second part of this evaluation, we provide summarized results for all tested servers, to illustrate the wide applicability of our attack methodology.

Note that we do not evaluate the expressiveness of code-reuse attacks based on Newton, i.e., we do not study whether Newton can produce Turing-complete attacks. The motivation behind this is that Turing-completeness neither guarantees nor is a prerequisite for successful exploitation and as such does not affect the applicability of Newton: an attacker is unlikely to care about finding all Turing-complete gadgets if only one or two already provide him with enough means to gain arbitrary code execution. We consider a study in which existing defenses are evaluated with respect to whether they prevent Turing-complete ROP attacks as an interesting starting point for future work.

Although our evaluation focuses on popular system services, the principles of Newton also apply to user applications like browsers, document readers, and word processors. The large memory footprint of such applications, however, means that our `libdft`-based DTA engine (which is 32-bit only) quickly runs out of memory. This limitation is not fundamental to Newton, and can be addressed in future work with additional engineering effort (i.e., porting `libdft` to x86_64).

6.1 In-Depth Analysis of nginx

We now evaluate the controllability of each executed indirect callsite in `nginx`, under all types of write and target constraints. We first examine the residual attack surface per target constraint, and then do the same for each write constraint.

Target constraints. Table 2 depicts the residual attack surface in `nginx` under different target constraints. We consider dynamic and static target constraints separately, in Tables 2a and 2b, respectively. It should be noted that the numbers shown for dynamic

Table 2: Number of permissible targets in `nginx` under each target constraint policy.

(a) Results for dynamic target constraints. ***Targets*****: absolute number of legal function targets found in the main `nginx` module, `libc`, other modules, and in total, respectively.** ***Target location*****: locations of the code pointers to legal targets (stack, heap, or `.data`/`.got`/other segment in a particular module).**

Dynamic target constraint	Targets				Target location										
	nginx	libc	other	total	stack	heap	nginx			libc			other		
							data	*GOT*	*other*	*data*	*GOT*	*other*	*data*	*GOT*	*other*
None	1035	2763	794	4592	–	–	–	–	–	–	–	–	–	–	–
Live +page	811	1264	411	2486	15	475	261	399	81	666	26	67	207	257	32
Computed	363	323	100	786	1	64	270	32	25	240	2	42	65	38	7
Live	362	316	89	767	1	64	269	31	25	237	2	41	60	31	6
Live ¬GOT	360	279	69	708	1	64	269	0	25	237	0	41	60	0	6

(b) Results for static target constraints. ***Targets*****: median (and minimum/maximum) number of legal function targets per callsite.** ***Target distribution*****: minimum/90th percentile/maximum number of targets pointing to each module, per callsite.**

Static target constraint	Targets (median)						Target distribution								
	nginx	libc	other	total	min	max	nginx			libc			other		
							min	*90p*	*max*	*min*	*90p*	*max*	*min*	*90p*	*max*
Bin types	328	960	370	1665	953	2820	201	758	758	549	1625	1625	203	437	437
Safe src types	117	176	65	376	2	394	2	135	153	0	230	230	0	69	72
Src types	12	0	0	19	1	61	1	58	61	0	0	20	0	0	28
Source	12	0	0	19	1	61	1	57	61	0	0	20	0	0	28

target constraints are susceptible to the coverage of our dynamic analysis. As mentioned, we assume a low-effort attacker; thus, the numbers shown in Table 2a cover the case where the attacker sends only a simple `GET` request to `nginx`. It is conceivable that a more determined attacker could uncover even more attack surface than shown in Table 2a.

Also note that we show absolute numbers for dynamic constraints, but median results for static constraints. This is because static target constraints limit the number of targets *per callsite*, while dynamic constraints limit the total number of legal pointers *in memory*.

To interpret the tables, we look at one example row from each table. We begin with an example from Table 2a. Consider the *Computed* target constraint, which is used by the *HCFI* defense subclass, implemented by Per-Input CFI [51]. Under this constraint, only code pointers which have been computed during program execution can be used by an attacker. Table 2a shows that after server initialization and handling of the `GET` request, 786 such pointers reside in memory. Thus, each indirect callsite may target each of these. Of the computed pointers, 1 was stored on the stack, and 64 on the heap. The remaining originate from the loaded modules: 270 from `nginx`' data sections (`.data`, `.data.rel.ro`, or `.rodata`), 32 from its global offset tables (`.got`, `.got.plt`), and 25 pointers were found in the remaining sections and other modules.

To explain Table 2b, we consider the *Safe src types* constraint, imposed by the *SafeIFCC/MCFI* defense subclass, which provides type-based caller/callee matching. In this, the median indirect callsite is allowed to target 176 `libc` functions, and 376 functions in total. The most restricted callsite may call only 2 functions, while the least restricted is allowed to call 394 functions. Each callsite may target at least 2 functions in `nginx`, while 90% of the callsites may target 69 functions in modules other than `nginx` or `libc`.

Overall, the main takeaway from Table 2 is the ease with which our methodology allows us to compare the strength of even extremely different defense subclasses. For instance, it is clear from Table 2a that the strongest dynamic target constraint is *Live ¬GOT*, imposed by the *XoM++* defense subclass. Comparing Tables 2a and 2b, it is also clear that static type-based constraints are in general stronger than dynamic ones, with the strongest target constraints being imposed by source-level type-based defenses. It is also worth noting that even for the strongest target constraints, there is still a significant residual attack surface.

Write constraints. We now consider the potential controllability of callsites in `nginx` given varying write constraints. Moreover, we also show that for each executed callsite, a nontrivial attack surface remains even under the strongest combinations of write and target constraints. To obtain information on which callsites are potentially controllable, we examine the taint information which Newton yields during the aforementioned attacker-initiated `GET` request to `nginx`. We present these results in Table 3.

To illustrate the semantics of Table 3, consider callsite number 27, at location `http_request.c:1126`. The target (*function pointer*) of this callsite is tainted by a code pointer, meaning that it can be controlled under write constraints which allow corrupting code pointers. Moreover, it is controllable from segregated state, so that the callsite is usable in a history flushing-based attack against *CsCFI*. All three arguments are tainted by non-pointer values, making them controllable even under the strictest write constraints. Controlling three arguments is often sufficient; for instance, both `execve` and `mprotect` take only three arguments (and `system` takes one).

Without any additional target constraints, the callsite at location `http_request.c:1126` has 4592 legal targets. Imposing the

Table 3: Taint information and residual attack surface for `nginx`. *Callsite*: controllable indirect call when sending a plain `HTTP GET` request. *Taint source*: taint information for the function pointer (target) and first six arguments (arguments actually used are underlined). None indicates an untainted value, while CPtr, DPtr, and ¬Ptr indicate taint through a code pointer, data pointer (and possibly CPtr), or non-pointer value (and possibly Ptr), respectively. *Segr*: marked if the call target is tainted by segregated state, and the call may thus be used in a history flushing attack against CsCFI. *Targets*: available targets for the given callsite under *Baseline* target constraints, *Live* ¬*GOT* (strongest dynamic) constraints, and *Source types* (strongest static) constraints. *Best*: available targets when combining *Live* ¬*GOT* and *Src types*.

	Callsite	Taint source								Targets			
		Func. ptr.	*Segr*	*Arg0*	*Arg1*	*Arg2*	*Arg3*	*Arg4*	*Arg5*	*Baseline*	*Live ¬GOT*	*Source types*	*Best*
1	ngx_connection.c:808	CPtr		¬Ptr	¬Ptr	¬Ptr	None	¬Ptr	None	4592	708	2	1
2	ngx_epoll_module.c:642	DPtr		DPtr	¬Ptr	None	None	None	None	4592	708	19	6
3	ngx_event.c:245	CPtr		None	¬Ptr	None	None	None	None	4592	708	1	1
4	ngx_event.c:286	CPtr		DPtr	¬Ptr	None	None	DPtr	DPtr	4592	708	2	2
5	ngx_event_accept.c:258	DPtr		DPtr	¬Ptr	¬Ptr	None	¬Ptr	None	4592	708	6	1
6	ngx_http_chunked_filter_module.c:79	CPtr		¬Ptr	None	None	None	None	¬Ptr	4592	708	58	18
7	ngx_http_chunked_filter_module.c:92	CPtr		¬Ptr	None	¬Ptr	None	¬Ptr	None	4592	708	11	8
8	ngx_http_charset_filter_module.c:235	CPtr		¬Ptr	¬Ptr	¬Ptr	None	None	None	4592	708	58	18
9	ngx_http_charset_filter_module.c:552	CPtr		¬Ptr	None	None	None	¬Ptr	¬Ptr	4592	708	11	8
10	ngx_http_core_module.c:852	¬Ptr		¬Ptr	¬Ptr	¬Ptr	None	None	¬Ptr	4592	708	8	7
11	ngx_http_core_module.c:874	CPtr		¬Ptr	¬Ptr	¬Ptr	None	None	¬Ptr	4592	708	58	18
12	ngx_http_core_module.c:906	¬Ptr		¬Ptr	¬Ptr	¬Ptr	None	¬Ptr	None	4592	708	58	18
13	ngx_http_core_module.c:1075	CPtr		¬Ptr	¬Ptr	¬Ptr	None	None	¬Ptr	4592	708	58	18
14	ngx_http_core_module.c:1357	¬Ptr		¬Ptr	¬Ptr	DPtr	None	¬Ptr	¬Ptr	4592	708	58	18
15	ngx_http_core_module.c:1825	CPtr		¬Ptr	None	None	None	None	¬Ptr	4592	708	58	18
16	ngx_http_core_module.c:1840	CPtr		¬Ptr	None	None	None	None	None	4592	708	11	8
17	ngx_http_gzip_filter_module.c:256	CPtr		¬Ptr	None	None	None	¬Ptr	None	4592	708	58	18
18	ngx_http_gzip_filter_module.c:323	CPtr		¬Ptr	None	None	None	None	None	4592	708	11	8
19	ngx_http_headers_filter_module.c:152	CPtr		¬Ptr	None	None	None	None	None	4592	708	58	18
20	ngx_http_log_module.c:252	DPtr		¬Ptr	¬Ptr	None	None	¬Ptr	¬Ptr	4592	708	6	1
21	ngx_http_log_module.c:297	DPtr		¬Ptr	¬Ptr	DPtr	None	¬Ptr	¬Ptr	4592	708	12	11
22	ngx_http_not_modified_filter_module.c:61	CPtr		¬Ptr	None	¬Ptr	None	None	¬Ptr	4592	708	58	18
23	ngx_http_postpone_filter_module.c:82	CPtr		¬Ptr	None	None	None	None	None	4592	708	11	8
24	ngx_http_range_filter_module.c:230	CPtr		¬Ptr	None	None	None	¬Ptr	None	4592	708	58	18
25	ngx_http_range_filter_module.c:551	CPtr		¬Ptr	None	¬Ptr	None	¬Ptr	None	4592	708	11	8
26	ngx_http_request.c:514	DPtr		DPtr	¬Ptr	¬Ptr	None	¬Ptr	None	4592	708	19	6
27	ngx_http_request.c:1126	CPtr	✓	¬Ptr	¬Ptr	¬Ptr	None	None	¬Ptr	4592	708	3	3
28	ngx_http_request.c:3002	¬Ptr		¬Ptr	¬Ptr	¬Ptr	None	¬Ptr	None	4592	708	58	18
29	ngx_http_ssi_filter_module.c:329	CPtr		¬Ptr	None	None	None	None	¬Ptr	4592	708	58	18
30	ngx_http_ssi_filter_module.c:392	CPtr		¬Ptr	None	None	None	¬Ptr	¬Ptr	4592	708	11	8
31	ngx_http_userid_filter_module.c:205	CPtr		¬Ptr	¬Ptr	None	None	None	¬Ptr	4592	708	58	18
32	ngx_http_variables.c:404	¬Ptr		¬Ptr	¬Ptr	¬Ptr	None	¬Ptr	¬Ptr	4592	708	61	49
33	ngx_http_write_filter_module.c:238	¬Ptr	✓	¬Ptr	¬Ptr	None	None	¬Ptr	None	4592	708	2	1
34	ngx_output_chain.c:65	¬Ptr		¬Ptr	None	None	None	None	None	4592	708	11	8
35	ngx_palloc.c:80	¬Ptr		¬Ptr	None	¬Ptr	None	None	¬Ptr	4592	708	56	7

strongest dynamic target constraint (*Live* ¬*GOT*) reduces this to 708 targets, while the strongest static target constraint (*Source types*) allows only 3 targets; the same number of targets as is allowed under the combination of these write constraints.

Note in Table 3 that 13 of the 35 callsites have a target that is tainted by a non-code pointer value, making them controllable even when code pointers are protected. Moreover, 8 callsites have a target tainted by a non-pointer value, making these callsites controllable under *all* write constraints imposed by current defenses. Many of these callsites have a significant number of legal targets, ranging up to 49 targets even when combining the strongest static and dynamic target constraints.

6.2 Generalized Results

Table 4 shows that **`nginx`** is representative for all evaluated servers. The fraction of tainted callsites is comparable, with the exception that callsites in **`httpd`** are not controllable using segregated state; **`httpd`** creates a new process for each connection, preventing our history flushing attack. In all evaluated servers, attacker-controlled callsites remain even under ¬Ptr write constraints.

Moreover, in all servers, a significant number of legal targets remain even under the strongest dynamic target constraints (*Live* ¬*GOT*), with the exception of a small number (the aforementioned cases with **`httpd`**, and one case in **`memcached`**). The same is true

Table 4: Summarized number of controllable callsites and targets for each server. *Callsites*: number of tainted (controllable) callsites under varying write constraints. *Targets (dynamic)*: total permissible targets (absolute) under each dynamic target constraint. *Targets (static)*: total permissible targets (median) under each static target constraint.

Server	Callsites		Targets (dynamic)					Targets (static)		
	Write constraint	*Tainted*	*Baseline*	*Live +page*	*Computed*	*Live*	*Live ¬GOT*	*Bin types*	*Safe src types*	*Src types*
	None	35						1,952	988	201
	¬CPtr	13						1,952	953	193
nginx	¬Ptr	8	4,592	2,336	786	767	708	1,952	787	160
	Segr	2						1,571	108	5
	Segr & ¬Ptr	1						1,571	2	2
	None	12						1,686	249	50
	¬CPtr	7						1,512	228	37
lighttpd	¬Ptr	2	4,450	1,867	497	474	409	1,187	56	6
	Segr	8						1,686	230	39
	Segr & ¬Ptr	2						1,187	56	6
	None	33						3,464	1,471	310
	¬CPtr	27	6,113	3,835	2,002	1,985	1,928	3,464	1,469	302
httpd	¬Ptr	13						3,408	1,079	139
	Segr	0	0	0	0	0	0	0	0	0
	Segr & ¬Ptr	0	0	0	0	0	0	0	0	0
	None	14						2,253	470	219
	¬CPtr	11						2,253	470	219
redis	¬Ptr	11	5,381	2,311	771	612	546	2,253	470	219
	Segr	2						1,227	13	11
	Segr & ¬Ptr	2						1,227	13	11
	None	8						2,314	275	35
	¬CPtr	3	4,326	2,420	752	738	391	1,624	243	7
memcached	¬Ptr	3						1,624	243	7
	Segr	1						2,105	18	18
	Segr & ¬Ptr	0	0	0	0	0	0	0	0	0
	None	43						2,762	1,323	393
	¬CPtr	40						2,762	1,253	383
bind	¬Ptr	39	7,693	2,829	1,028	1,010	918	2,762	1,241	371
	Segr	1						1,936	199	20
	Segr & ¬Ptr	1						1,936	199	20

for static target constraints; even under source-level type-based target constraints, an attacker has multiple targets to choose from (ranging from 2 to 393 targets) in each case where callsite corruption is possible. For several servers, including **`nginx`**, **`lighttpd`**, **`redis`**, and **`bind`**, these results apply even to a segregated state attack model with type-based target constraints.

These results show that Newton is capable of locating controllable callsites and a choice of potential targets under even the strongest defenses. Recall that these results assume a low-effort attacker, sending only a single request to each server; thus, these results are a lower-bound for the number of controllable callsites.

7 CONSTRUCTING ATTACKS WITH Newton

This section documents our experience using Newton to bypass two advanced state-of-the-art defenses: CsCFI [16, 28, 44, 54, 67] and CPI [43]. Our case studies are constructed in an architecture-independent fashion: unlike traditional ROP, we operate on program semantics. Thus, our results are generally applicable on different architectures, such as x86 and ARM. We specifically focus our analysis on ideal implementations of CPI and CsCFI, given that existing work has already discussed the general limitations of CFI [13, 14, 23, 27, 32, 33] and leakage-resistant randomization [58].

7.1 CsCFI

In this case study, we target CsCFI on **`nginx`**. As described in §5, to bypass CsCFI's write constraint (*Segr*), we look for callsites controllable from a segregated (connection-specific) state. We (1) open a connection c_1 to prepare its memory state, (2) flush the branch history by sending n parallel requests over another connection c_2 (disabling CsCFI's protection), and finally (3) send a request over connection c_1 to divert control flow from a $C1$-controlled callsite.

As shown in Table 3, Newton provides us with two candidate callsites to bypass CsCFI (those with the *Segr* column checked). We select callsite 33 in the function `ngx_http_write_filter`:

```
chain = c->send_chain(c, r->out, limit);
```

Here, `c` is a pointer to our connection state (`ngx_connection_t`), which contains a code pointer called `send_chain`. Clearly, the connection state and code pointers stored therein are isolated from other connections. Other than `send_chain` and `c` itself (first argument), Newton also reports that the second `r->out` argument is tainted and controllable from corrupted connection-specific state.

With manual inspection, we verified that (1) controlling the target and arguments with an arbitrary memory write to segregated state allows request handling to complete without crashes, (2) we also control the third argument by controlling the `limit_rate` and `limit_rate_after` configuration variables and flipping a single (uncovered) branch in the execution, and (3) execution continues correctly if the `send_chain` call is diverted to a different target returning a 0 value, allowing us to chain successful calls via repeated interactions with the server.

Newton also provides us with a list of all the possible 4592 targets (no target constraints) for our selected callsite. We target `mprotect` to escalate code reuse to a code corruption attack. This function expects three arguments: (1) the start address of the affected memory region, (2) the size of the region, and (3) the protection flags.

To select the start address, we overwrite the `c` pointer and repoint it to a counterfeit object prepared with identical connection state in a memory location of our choosing. To select the protection flags, we overwrite the `limit_rate_after` variable to ensure the final `limit` computation has the `PROT_READ|WRITE|EXEC` bits set in the lowest byte. To select the size, we need to redirect the `r->out` pointer to a value of our choosing. However, it is challenging to enforce a small `r->out` pointer value, since the lower part of the address space is not normally mapped. To address this challenge, we aim for a large `mprotect` surface, spanning from the heap (i.e., the controlled `c` pointer) all the way to `libc` code. The latter is the next region in line in the address space, only separated from the heap by a single unmapped gap. To fill the gap, we use a preliminary request to redirect control to libc's `malloc` without worrying about its argument—since this is a pointer, calling `malloc` will result in a large allocation, adjacent to `libc` in our setting.

At this point, we safely hijack our victim callsite to call `mprotect` and make the (now larger) heap and the entire `libc` code readable, writeable, and executable. Once `mprotect` succeeds, we issue another request to corrupt `libc`'s `gettimeofday` function with our own shellcode. The shellcode runs when `nginx` processes the next request, giving us arbitrary code execution. Figure 4 shows an overview of the attack.

Evidently, even a state-of-the-art defense like CsCFI alone is not sufficient to stop an attacker armed with dynamic analysis. Instead, to limit the power of these attacks, we must carefully combine context-sensitive CFI with traditional CFI or other defenses. Note that state-of-the-art binary-level CFI policies based on argument/return count matching (TypeArmor) cannot prevent our `mprotect` hijacking attack, given that the callsite is diverted with a compatible function signature. Thus, stronger static (e.g., *Src types*) or dynamic (e.g., *Live*) target or write constraints that protect pointer corruption (e.g., CPI's ¬*Ptr*) are necessary.

To confirm the real-world applicability of Newton, we successfully implemented the above attack in practice. Using `gdb` to mimic an attacker's arbitrary read and write memory primitive, we recorded a video that shows how one can use our attack to mark `libc` memory pages as readable, writable, *and* executable. The video, accompanied with annoted details, is available on our project webpage.[1]

7.2 CPI

In this case study, we target CPI on `nginx`. CPI enforces a ¬Ptr write constraint, protecting code and data pointers. Thus, we use Newton's results in Table 3 to find callsites tainted by a non-pointer value, and select callsite 32. The callsite is in the function `ngx_http_get_indexed_variable`, and selects its callee from an array of structures with function pointers, as follows:

```
v[index].get_handler(r, &r->variables[index],
                     v[index].data)
```

Newton's output pinpoints the taint source that we need to corrupt to control the `get_handler` function pointer: the `data` field in an `ngx_http_log_op_s` structure. It is worth noting how little effort it takes to find this dependency with Newton, as inspecting the source code reveals a complex data flow. The tainted data flows through multiple `nginx`-specific data structures and functions—none of which our low-effort attacker needs to know.

Newton also reveals that the taint source for the three arguments (Table 3) are all tainted by a non-pointer value. The last argument is controllable via the tainted `index`. The first two arguments are controllable by corrupting the allocator state much earlier in the execution. For example, the taint of the first `ngx_http_request_t*` argument originates 11 functions earlier in the execution, in `ngx_http_process_request_headers`. Again, Newton hides this complexity from the user.

With simple manual inspection, we also found that (1) the request data pointed to by the first argument is controllable by sending an incomplete `HTTP` request (which we complete later to trigger the exploit), (2) controlling the target and arguments with an arbitrary memory write allows request handling to complete without crashes, and (3) execution continues if the `get_handler` call is diverted to a different target, making it possible to chain calls via repeated interactions with the server.

Other than information on how to effect an arbitrary memory write and divert control flow, Newton also provides us with a list of the 767 usable targets stored in memory. This reflects CPI's *Live* target constraint. A complication is that we only control the `index` into the `v` array of `ngx_http_variable_t` structures. Since each structure contains 6 word-sized fields, only 1/6 of memory can be used to select live code pointer targets. Fortunately, this alignment restriction is bypassable using memory massaging (on the heap, stack, etc.) [11]. Moreover, Newton found the address of `dlopen` live in memory, allowing us to load arbitrary shared objects on the victim system and expand the set of available live targets.

For example, if we call `dlopen` on `"/bin/ed"` or other shared objects which use the `system` library call, we force the linker to bind the `system` code pointer in memory (GOT). This is easier after corrupting the linker configuration (`LD_LIBRARY_PATH`, `LD_BIND_NOW`). At that point, we again corrupt the `index` integer to redirect `get_handler` to the newly created live code pointer.

[1] https://vusec.net/projects/newton

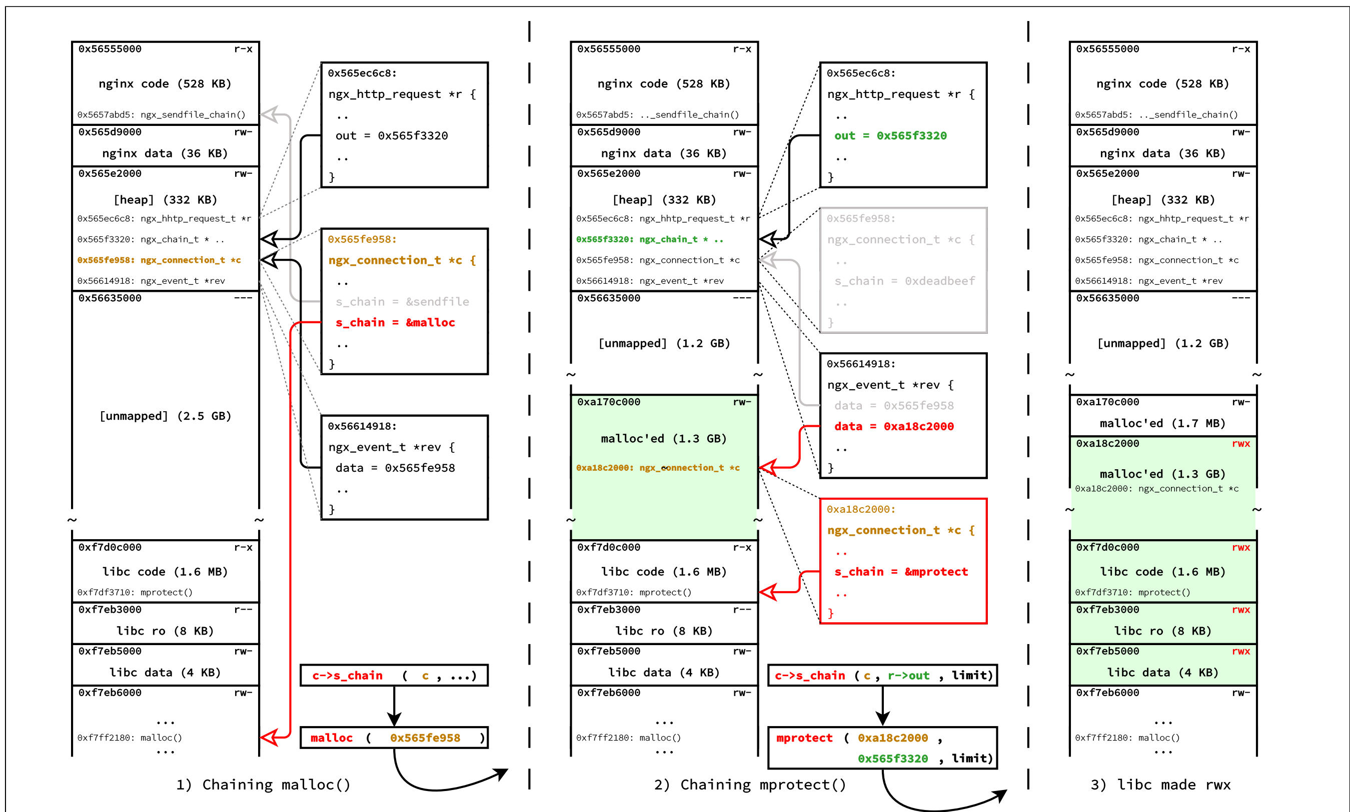

Figure 4: Chaining `malloc` and `mprotect` in `nginx` to make `libc` code pages writable, using the callsite `c->send_chain(c, r->out, limit)`. This figure illustrates memory layout and key variables of the `nginx` process before, during, and after our attack against CsCFI.
We first overwrite the **`send_chain`** code pointer in **`c`** with the address of **`malloc`**. Since the callsite uses the address of **`c`** as first argument, this results in a **`0x565fe958`** B = 1.3 GB allocation, adjacent to **`libc`** code. We then overwrite the same code pointer with the address of **`mprotect`** and construct a counterfeit **`c`** structure at a convenient location: knowing that the value of **`r->out`** will be the **`len`** argument for **`mprotect(void *addr, size_t len, int prot)`**, we place **`c`** at **`libc`** − **`r->out`**, i.e., **`0xf7eb6000`** − **`0x565f3320`** = **`0xa18c2000`** (rounded to the page boundary). To make **`nginx`** use our counterfeit object, we must also update the **`data`** pointer in the relevant **`ngx_event_t *rev`** structure. By using partial HTTP requests, we divide a single control-flow diversion into multiple steps: (1) open a connection c_1 and send a partial request; (2) use the arbitrary memory read/write primitive to corrupt the connection state of c_1, e.g., overwrite the **`send_chain`** code pointer; (3) open connections $c_2 \ldots c_n$ to perform n HTTP requests in parallel to flush CsCFI history, i.e., recorded branches that set **`send_chain`** to **`ngx_sendfile_chain`** are pruned from memory, (4) finish the partial request of c_1, triggering the control-flow diversion while CsCFI is unable to find in which context the overwritten code pointer was originally set.

Subsequently, we send another request to chain an invocation of the (now live) **`system`** library call, allowing us to execute arbitrary commands on the victim system. To "massage" the GOT to obtain a correctly aligned **`system`** code pointer, we carefully choose the **`system`**-dependent shared object to load.

We note that, other than CPI, the above attack bypasses all the defenses in the bottom-left quadrant marked by the <¬Ptr, *Live*> data point in Figure 2, including CCFI, TASR, PtrRR, XoM, and TypeArmor. Thus, an important lesson learned is that we must combine CPI with other strong defenses to further limit the attack surface. CPI combined with a secure implementation of CsCFI, for instance, would prevent us from controlling callsite 32.

When crafting the above attack in practice, we found that GNU **`libc`** enforces strict constraints on the flags provided to **`dlopen`**: unused bits should be zero, or else an error is returned.[2] This limits our attack, as it means that **`index`** should now be chosen such that the address of **`r->variables[index]`** is a valid flag for **`dlopen`** (e.g., `RTLD_NOW`), while **`v[index].get_handler`** still points to **`dlopen`**. Successful exploitation thus depends on the **`libc`** version. Musl **`libc`**, for example, does not enforce these constraints. Running **`nginx`** with musl **`libc`**, however, voids **`dlopen`** pointers in memory. Instead, we found code pointers to many functions of the **`exec()`** family, opening alternative ways for bypassing CPI.

[2] https://sourceware.org/git/?p=glibc.git;h=3e539cb47e9fabfdda295926b4270b0f...

8 RELATED WORK

As we already discussed code-reuse defenses at length in the paper, this section discusses the literature on code-reuse attacks only.

Return-into-libc (ret2libc) [25] represents the first generation of code-reuse attacks. Traditionally targeting the 32-bit x86 ISA, ret2libc uses a memory corruption vulnerability to inject a return address on the stack pointing to an existing (`libc`) function, followed by function arguments. Thus, a subsequent `ret` instruction transfers control to the prepared function, essentially thwarting DEP [2]. By preparing multiple call frames, function calls can be chained. On the x86-64 architecture, most function arguments are passed in CPU registers, making ret2libc more challenging.

Return-Oriented Programming (ROP) [57] generalizes ret2libc, and is now the de-facto standard in real-world code-reuse attacks. ROP also manipulates the stack, but doesn't chain complete functions. Instead, ROP uses small code fragments ending in return instructions, called *gadgets*. ROP is an extremely potent attack technique, which allows attackers to implement arbitrary Turing-complete computations in most practical programs [60].

The initial ROP attack signaled the start of an arms race around a third-generation of code-reuse attacks. Several defense techniques were developed, only to be shown susceptible to improved code-reuse attacks. *Jump-Oriented Programming* (JOP) [9] bypasses some execution monitoring defenses [24] and *Counterfeit Object-Oriented Programming* (COOP) [59] and related attacks [13, 14, 23, 27, 32, 33] bypass many existing Control-Flow Integrity (CFI) [1]-based defenses. Finally, other attacks such as JIT ROP [22, 63], SROP [10], and AOCR [58] bypass information hiding defenses, including leakage-resistant variants [58]. The "gadget-stitching" model extends even beyond code reuse, also adopted by state-of-the-art techniques to craft data-only attacks [37, 38]. Note that although these recent efforts on Data-Oriented Programming (DOP) show similar weaknesses in modern defenses as outlined in this paper, a key difference is that most of those defenses were never designed to mitigate data-only attacks. Attacks crafted with Newton, on the other hand, fall within the defenses' threat models.

Although the way Newton finds gadgets shows some similarity to how ACICS gadgets are found [27], the latter are more constrained: only attacks where the function pointer and arguments are directly corruptible on the heap or in global memory are considered. As shown in §7, Newton finds more sophisticated attacks, where these elements may be corrupted in complex, indirect ways.

The focus on (manual or automatic) static analysis makes code reuse increasingly complex given increasingly sophisticated defenses. With Newton, we show that a switch to a simple and natural dynamic analysis approach significantly simplifies the discovery and stitching of gadgets, even in the face of state-of-the-art defenses. Moreover, we argue that ret2libc-style attacks on 64-bit architectures are not only practical, but also much easier, if an attacker piggybacks on the benign data flows of the application.

9 CONCLUSION

The "*geometry*" of innocent flesh on the bone has characterized ten years of code-reuse research: an attacker statically analyzes binary code to find gadgets, chains them together, and "calls" into security-sensitive syscalls. This model is simple to understand, but scales poorly as we assume increasingly sophisticated defenses.

In this paper, we showed that, by also considering the "*dynamics*" of innocent flesh on the bone, even a low-effort attacker can easily find useful defense-aware gadgets to craft practical attacks. We implemented Newton, a gadget-discovery framework based on simple static and dynamic (taint) analysis. Using Newton, we found gadgets compatible with state-of-the-art defenses in many real-world programs. We also presented an `nginx` case study, showing that a Newton-armed attacker can find useful gadgets and craft attacks that comply with the restrictions of strong defenses such as CPI and context-sensitive CFI.

Our effort ultimately shows that, to sufficiently reduce the attack surface against a dynamic attack model, we must combine multiple state-of-the-art code-reuse defenses or, alternatively, deploy more heavyweight defenses at the cost of higher overhead.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and input to improve the paper. This work was supported by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO CSI-DHS 628.001.021, and by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *CCS*.
[2] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. (2004). http://technet.microsoft.com/en-us/library/bb457155.aspx.
[3] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *CCS*.
[4] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *USENIX SEC*.
[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX SEC*.
[6] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *USENIX SEC*.
[7] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *CCS*.
[8] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *S&P*.
[9] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ASIACCS*.
[10] Erik Bosman and Herbert Bos. 2014. Framing Signals—A Return to Portable Shellcode. In *S&P*.
[11] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.
[12] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*.
[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX SEC*.
[14] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX SEC*.
[15] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *EuroS&P*.
[16] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*.
[17] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi.

2016. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In *PETS*.
[18] Stephen Crane, Andrei Homescu, and Per Larsen. 2016. Code Randomization: Haven't We Solved This Problem Yet?. In *SecDev*.
[19] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stephan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *S&P*.
[20] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *CCS*.
[21] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *ASIACCS*.
[22] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*.
[23] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX SEC*.
[24] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks. In *ACM STC*.
[25] Solar Designer. Return-to-libc attack. BugTraq. (Aug. 1997).
[26] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *S&P*.
[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin C. Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *CCS*.
[28] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *ASPLOS*.
[29] Jason Gionta, William Enck, and Per Larsen. 2016. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *CNS*.
[30] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *CODASPY*.
[31] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX SEC*.
[32] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *S&P*.
[33] Enes Goktas, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX SEC*.
[34] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *CODASPY*.
[35] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'D My Gadgets Go?. In *S&P*.
[36] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided Automated Software Diversity. In *CGO*.
[37] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *USENIX SEC*.
[38] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *S&P*.
[39] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *VEE*.
[40] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC*.
[41] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-Variant Execution Using Hardware-assisted Process Virtualization. In *DSN*.
[42] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the Gadgets: Binary-level Code Randomization Using Instruction Displacement. In *ASIACCS*.
[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI*.
[44] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *HPCA*.
[45] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to make ASLR win the Clone Wars: Runtime Re-Randomization. In *NDSS*.
[46] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *CCS*.
[47] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *CCS*.
[48] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*.
[49] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*.
[50] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *PLDI*.
[51] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *CCS*.
[52] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX SEC*.
[53] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *S&P*.
[54] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX SEC*.
[55] PaX Team. Address Space Layout Randomization (ASLR). (2003). pax.grsecurity.net/docs/aslr.txt.
[56] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *EuroSys*.
[57] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *TISSEC* 15, 1 (2012).
[58] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *NDSS*.
[59] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*.
[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *USENIX SEC*.
[61] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *CCS*.
[62] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *CCS*.
[63] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *S&P*.
[64] Mingshen Sun, John C. S. Lui, and Yajin Zhou. 2016. Blender: Self-Randomizing Address Space Layout for Android Apps. In *RAID*.
[65] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *CCS*.
[66] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX SEC*.
[67] Victor van der Veen, Dennis Andriesse, Enes Goktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-sensitive CFI. In *CCS*.
[68] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *S&P*.
[69] Stijn Volckaert, Bart Coppens, and Bjorn de Sutter. 2015. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. In *TDSC*.
[70] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. 2017. ReRanz: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *VEE*.
[71] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *CCS*.
[72] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *ASIACCS*.
[73] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*.
[74] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *S&P*.
[75] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX SEC*.