Q: Exploit Hardening Made Easy

Edward J. Schwartz, Thanassis Avgerinos and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{*edmcman, thanassis, dbrumley*}*@cmu.edu*

Abstract

Prior work has shown that return oriented programming (ROP) can be used to bypass W⊕X, a software defense that stops shellcode, by reusing instructions from large libraries such as libc. Modern operating systems have since enabled address randomization (ASLR), which randomizes the location of libc, making these techniques unusable in practice. However, modern ASLR implementations leave smaller amounts of executable code unrandomized and it has been unclear whether an attacker can use these small code fragments to construct payloads in the general case.

In this paper, we show defenses as currently deployed can be bypassed with new techniques for automatically creating ROP payloads from small amounts of unrandomized code. We propose using semantic program verification techniques for identifying the functionality of gadgets, and design a ROP compiler that is resistant to missing gadget types. To demonstrate our techniques, we build Q, an end-to-end system that automatically generates ROP payloads for a given binary. Q can produce payloads for 80% of Linux `/usr/bin` programs larger than 20KB. We also show that Q can automatically perform *exploit hardening*: given an exploit that crashes with defenses on, Q outputs an exploit that bypasses both W⊕X and ASLR. We show that Q can harden nine real-world Linux and Windows exploits, enabling an attacker to automatically bypass defenses as deployed by industry for those programs.

1 Introduction

Control flow hijack vulnerabilities are extremely dangerous. In essence, they allow the attacker to hijack the intended control flow of a program and instead execute whatever actions the attacker chooses. These actions could be to spawn a remote shell to control the program, to install malware, or to exfiltrate sensitive information stored by the program.

Luckily, modern OSes now employ W⊕X and ASLR together — two defenses intended to thwart control flow hijacks. Write xor eXecute (W⊕X, also known as DEP) prevents an attacker's payload itself from being directly executed. Address space layout randomization (ASLR) prevents an attacker from utilizing structures within the application itself as a payload by randomizing the addresses of program segments. These two defenses, when used together, make control flow hijack vulnerabilities difficult to exploit.

However, ASLR and W⊕X are not enforced completely on modern OSes such as OS X, Linux, and Windows. By completely, we mean enforced such that no portion of code is unrandomized for ASLR, and that injected code can never be executed by W⊕X. For example, Linux does not randomize the program image, OS X does not randomize the stack or heap, and Windows requires third party applications to explicitly opt-in to ASLR and W⊕X. Enforcing ASLR and W⊕X completely does not come without cost; it may break some applications, and introduce a performance penalty.

Previous work [41] has shown that systems that do not randomize large libraries like libc are vulnerable to return oriented programming (ROP) attacks. At a high level, ROP reuses instruction sequences already present in memory that end with `ret` instructions, called *gadgets*. Shacham showed that it was possible to build a Turing-complete set of gadgets using the program code of libc. Finding ROP gadgets has since been, to a large extent, automated when large amounts of code are left unrandomized [16, 21, 38]. However, it has been left as an open question whether current defenses, which randomize large libraries like libc but leave small amounts of code

unrandomized, are sufficient for all practical purposes, or permit such attacks.

In this paper, we show that current implementations are vulnerable by developing automated ROP techniques that bypass current defenses and work even when there is only a small amount of unrandomized code. While it has long been known that ASLR and W⊕X offer important protection in theory, our main message is that current practical implementations make compatibility and performance tradeoffs, and as a result it is possible to *automatically harden existing exploits* to bypass these defenses.

Bypassing defenses on modern operating systems requires ROP techniques that work with whatever unrandomized code is available, and not just pre-determined code or large libraries. To this end, we introduce several new ideas to scale ROP to small code bases.

One key idea is to use semantic definitions to determine the function, if any, of an instruction sequence. For instance, rather than defining `movl *, *; ret` as a move gadget [21, 38], we use the semantic definition `OutReg` ← `InReg`. This allows us to find unexpected gadgets such as realizing `imul $1, %eax, %ebx; ret`[1] is actually a move gadget.

Another key point is that our system needs to gracefully handle missing gadget types. This is comparable to writing a compiler for an instruction set architecture, except with some key instructions removed; the compiler must still be able to add two numbers even when the `add` instruction is missing. We use an algorithm that searches over many combinations of gadget types in such a way that will synthesize a working payload even when the most natural gadget type is unavailable. Prior work [16, 21, 38] focuses on finding gadgets for all gadget types, such that a compiler can then create a program using these gadget types. This direct approach will not work without additional logic if some gadget types are missing. However, we are not aware of prior work that considers this. This is essential in our application domain, since most programs will be missing some gadget types.

Our results build on existing ROP research. Previous ROP research was either performed by hand [6, 9, 41], or focused on large code bases such as libc [38] (1,300KB), a kernel [21] (5,910KB) or mobile libraries [16, 24] (size varies; on order of 1,000KB). In contrast, our techniques work on small amounts of code (20KB). In our evaluation (Section 7), we show that Q can build ROP payloads for 80% of Linux programs larger than 20KB. Q can also transplant the ROP payloads into an existing exploit that does not bypass defenses, effectively hardening the original exploit to bypass W⊕X and ASLR. Recent work in automatic exploit generation [2, 5] can be used to generate such exploits. We show that Q can automatically harden nine exploits for real binary programs on Linux and Windows to bypass implemented defenses. Since these defenses can automatically be bypassed, we conclude that they provide insufficient security.

[1] We use AT&T assembly syntax in this paper, i.e., the source operand comes first.

Contributions. Our main contribution is demonstrating that existing ASLR and W⊕X implementations do not provide adequate protection by developing automated techniques to bypass them. First, we perform a survey of modern implementations and show that they often do not protect all code even when they are "turned on". This motivates our problem setting. Second, we develop ROP techniques for small, unrandomized code bases as found in most practical exploit settings. Our ROP techniques can automatically compile programs written in a high-level language down to ROP payloads. Third, we evaluate our techniques in an end-to-end system, and show that we can automatically bypass existing defenses for nine real-life vulnerabilities on both Windows and Linux.

2 Background and Defense Survey

There is a notion that code reuse attacks like return oriented programming are not possible when ASLR is enabled at the system level. This is only half true. If ASLR is applied to all program segments, then code reuse is intuitively difficult, since the attacker does not know where any particular instruction sequence will be in memory. However, ASLR is not currently applied to all program segments, and we will show that attackers can use this to their advantage. In this section, we explain the W⊕X and ASLR defenses in more detail, focusing on when a program segment may be left unprotected.

Table 1 summarizes some of these limitations. The key insight that we make use of in this paper is that program images are always unrandomized unless the program explicitly opts in to randomization. On Linux, for instance, this mean that developers must set non-default compiler flags to enable randomization. Another surprise is that W⊕X is often disabled when older hardware is used; some virtualization platforms by default will omit the virtual hardware needed to enable W⊕X.

2.1 W⊕X

W⊕X prevents attackers from injecting their own payload and executing it by ensuring that protected program segments are not writable and executable at the same time

Operating System	W⊕X	ASLR		
		stack, heap	libraries	program image
Ubuntu 10.04	Yes	Yes	Yes	Opt-In
Debian Sarge	HW	Yes	Yes	Opt-In
Windows Vista, 7	HW	Yes	Opt-In	Opt-In
Mac OS X 10.6	HW	No	Yes	No

Table 1: Comparison of defenses on modern operating systems for the x86 architecture with default settings. Opt-In means that programs and libraries must be explicitly marked by the developer at compile time for the protection to be enabled, and that some compilers do not enable the marking by default. HW denotes that the level of protection depends on hardware.

(Writable ⊕ eXecutable[2]). Attackers have traditionally included shellcode (executable machine code) in their exploits as payloads. Since shellcode must be written to memory at runtime, it cannot be executed because of the W⊕X property.

W⊕X Implementation W⊕X is implemented [29, 30, 35] using a `NX` (no execute) bit that the hardware platform enforces: if execution moves to a page with the `NX` bit enabled, the hardware raises a fault. On x86, this bit can be set using the PAE addressing mode [22].

PAE support is disabled by default in Ubuntu Linux, since some older hardware does not support it. The ExecShield [31] patch, which is included in Ubuntu, can emulate W⊕X by using x86 segments, even when hardware `NX` support is not available. Other distributions (such as Debian) do not include the ExecShield patch, and do not provide any W⊕X protection in default kernels.

Windows 7 enables W⊕X[3] by default for processors supporting the `NX` bit. However, it only enforces W⊕X for binaries and libraries marked as W⊕X compatible. Many notable third-party software programs such as Oracle's Java JRE, Apple Quicktime, VLC Media Player and others do not opt-in to W⊕X [36].

Limitations The main limitation of W⊕X is that it only prevents an attacker from utilizing *new* payload code. The attacker can still reuse existing code in memory. For instance, an attacker can call `system` by launching a return-to-libc attack, in which the attacker creates an exploit that will call a function in libc without injecting any shellcode. W⊕X does not prevent return-to-libc attacks because the executed code is in libc and is intended to be executable at compile time. Return Oriented Programming is another, more advanced attack on W⊕X, which we discuss in Section 2.3.

[2]W⊕X is actually a misnomer, because memory is allowed to be unwritable and non-executable, but $0 \oplus 0 = 0$.

[3]W⊕X is called DEP by the Windows community. Windows also contains *software DEP*, but this is unrelated to W⊕X [30].

2.2 ASLR

ASLR prevents an attacker from directly referring to objects in memory by randomizing their locations. This stops an attacker from being able to transfer control to his shellcode by hardcoding its address in his exploit. Likewise, it makes return-to-libc and ROP using libc difficult, because the attacker will not know where libc is located in memory.

Implementation ASLR implementations randomize some subset of the stack, heap, shared libraries (e.g., libc), and program image (e.g., the `.text` section).

Linux [31, 34] randomizes the stack, heap, and shared libraries, but not the program image. Programs can be manually compiled into position independent executables (PIEs) which can then be loaded to multiple positions in memory. Modern distributions [14, 44] only compile a select group of programs as PIEs, because doing so introduces a performance overhead at runtime.

Windows Vista and 7 [29, 43] can randomize the locations of the program image, stack, heap, and libraries, but only when the program and all of its libraries opt-in to ASLR. If they do not, some code is left unrandomized. Many third-party applications including Oracle's Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked as ASLR compatible [36]. Ultimately, this means most Windows binaries have unrandomized code.

Limitations Some attacks on ASLR implementations take advantage of the low entropy available for randomization. For instance, Shacham, et al. [42] show that brute forcing ASLR on a 32-bit platform takes about 200 seconds on average. (We do not consider attacks that take more than one attempt in this paper; we create exploits that succeed on the first try.) Other attacks, such as `ret2reg` attacks, allow the attacker to transfer control to their payload by utilizing pointers leaked in registers or memory [32]. For instance, the `strcpy` function returns such a pointer to the destination string in the %eax register. The applicability of these attacks are heavily dependent on the vulnerable program.

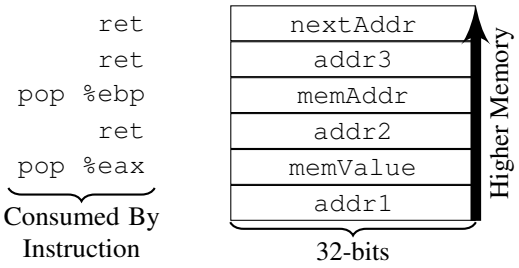

Figure 1: Example payload for storing `memValue` to `memAddr` for the scenario described in the text. This payload will transfer control to address `nextAddr` after writing to memory.

2.3 Return Oriented Programming

Return Oriented Programming is a generalization of the return-to-libc attack. In a return-to-libc attack the attacker reuses entire functions from libc. With ROP, the attacker uses instruction sequences found in memory, called gadgets, and chains them together. ROP attacks are desirable because they allow the attacker to perform computations beyond the functions of libc (or whatever code is unrandomized). This is especially important in the context of modern systems, because the unrandomized code may not contain useful funcions for the attacker. Researchers [16, 21, 41] have shown that it is possible to find gadgets for performing Turing-complete operations in libc, the windows kernel, and mobile phone libraries.

Example 2.1 (Return Oriented Programming)**.** Assume that the following instruction sequences are in memory at `addr1`: `pop %eax; ret`; at `addr2`: `pop %ebp; ret`; and at `addr3`: `movl %eax, (%ebp); ret`. The first two sequences pop a 32-bit value from the stack, store it into a register, and then jump to the address stored on the stack. If the attacker controls the stack and can cause one of these instruction sequences to execute, then the attacker can put values in %eax and %ebp and transfer control to another address. By chaining together all three instruction sequences, the attacker can write to memory (and still transfer control to the next gadget). The attacker's payload for writing `memValue` to `memAddr` is shown in Figure 1. It is possible to execute arbitrary programs by stringing together gadgets of different types.

3 System Overview

In the next two sections, we describe Q[4], our system for automatic exploit hardening. Figure 2 shows the end-to-end workflow of Q, which is divided into two phases. The first phase automatically generates ROP payloads (Section 4). The second phase is exploit hardening (Section 5). In exploit hardening, Q takes the ROP payloads generated in the first stage and transplants them into existing exploits which do not bypass defenses. The resulting exploit can then bypass W⊕X and ASLR.

4 Automatically Generating Return-Oriented Payloads

Q's end-to-end return oriented programming system consists of a number of different stages. Previous research on automated ROP has typically focused on one specific stage; for instance, gadget discovery [16, 24, 38] or compilation [6]. Since Q is an end-to-end ROP system, it has multiple stages. We describe each stage in the context of a user's potential interaction with the system below.

4.1 Example Usage Scenario

Assume that Alice wants to create a ROP payload that calls `system` (her target program) using instructions from `rsync`'s unrandomized code (her source program). Here, source program means the program from which Q takes instruction sequences to construct gadgets (e.g., the program with a vulnerability), and target program means the program Alice wants to run (using ROP). Alice would use the following stages of Q, which are depicted in the top half of Figure 2:

Gadget Discovery The first stage of Q is to find gadgets in the source program that Alice provides — in this case, `rsync`. The gadgets will be the building blocks for the ROP payloads that are ultimately created, and thus it is important to find as many as possible. Q finds gadgets of various types (specified in Table 2) by using semantic program verification techniques on the instruction sequences found in `rsync`.

Q's semantic engine allows it to find gadgets that humans might miss. For instance, Q can automatically determine that `lea (%ebx,%ecx,1),%eax; ret` adds %ebx with %ecx and stores the result in %eax. Likewise it discovers that `sbb %eax, %eax; neg %eax; ret` moves the carry flag (CF) to %eax.

[4] We name our system after Q from the James Bond movies, who creates, modifies, and combines gadgets to help Bond meet his objectives.

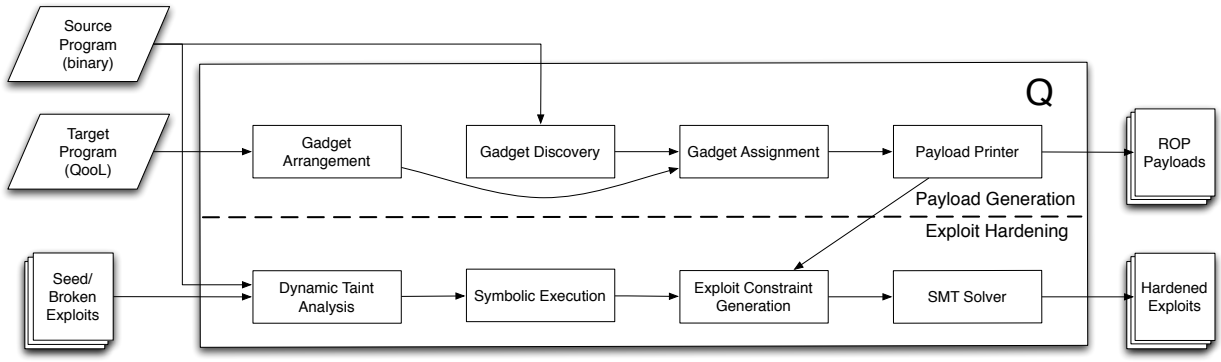

Figure 2: An overview of Q's design.

Input Alice writes the target program that she wants to execute in Q's high level language, QooL (shown in Table 3). The target program calls `system` with the desired arguments (e.g., `/bin/sh`).

Gadget Arrangement Q builds a list of *gadget arrangements*. Each gadget arrangement is a way of implementing the target program using different types of gadgets. For example, we show a gadget arrangement for writing to memory in Figure 3; this arrangement is the most natural way of storing to memory, but will not work if Q can not find a StoreMemG gadget. Gadget arrangement is somewhat analogous to instruction selection in a compiler. A major difference is that a regular compiler can use whichever instructions it chooses, but Q is limited to the gadget types that were found during gadget discovery.

Gadget arrangement allows Q to cope with missing gadgets. If the most natural choice of gadget is not available, Q effectively tries to synthesize a combination of other gadgets that will have the same semantics. We are not aware of other ROP compilers that consider this.

Gadget Assignment Gadget assignment takes gadgets found during discovery, and assigns them in the arrangements that Q generated. The difficulty is that assignments must be compatible. This means that the output register of one gadget must match the input register on the receiving gadget. Likewise, gadgets cannot clobber a register if that value is waiting to be used by a future gadget. This phase is roughly analogous to register allocation in a traditional compiler. Unlike a traditional compiler, Q cannot spill registers to memory, since this usually increases register pressure instead of decreasing it. As an example, Q assigns the following gadgets from `rsync` to implement the gadget arrangement in Figure 3:

```
; Load value into %eax
pop %ebp; ret; xchg %eax, %ebp; ret
; Load address-0x14 into %ebx
pop %ebx; pop %ebp; ret
; Store memory
mov %eax, 0x14(%ebx); ret
```

Output Finally, as long as at least one of the gadget arrangements has been assigned compatible gadgets, Q prints out payload bytes that Alice can use in her exploit. If Alice already has an exploit that no longer works because of W⊕X and ASLR, she can feed in the generated ROP payload along with her old exploit to the second phase of Q (see Section 5) to harden her exploit against these defenses.

We now explain each stage of Q in more detail.

4.2 Gadget Discovery

Not every instruction sequence can be used as a gadget. Q requires each gadget to satisfy four properties:

Functional Each gadget has a *type* (from Table 2) that defines its function. In our system, a gadget's type is specified *semantically* by a boolean predicate that must always be true after executing the gadget.

Control Preserving Each gadget must be capable of transferring control to another gadget. In our system, this means that the gadget must end with `ret` or some semantically equivalent instruction sequence (e.g., `pop %eax; jmp *%eax`).

Known Side-effects The gadget must not have unknown side-effects. For instance, the gadget must not write to any undesired memory locations.

Constant Stack Offset Most gadget types require the stack pointer to increase by a constant offset after each execution.

Although we found these requirements to work well, we discuss alternatives to the control preservation and

known side-effects requirements in Section 8.

4.2.1 Gadget Types

The set of gadget types in Q defines a new instruction set architecture (ISA) in which each gadget type functions as an instruction. At a high-level, we specify the meaning of each gadget type with a postcondition $\mathcal{B}$ that must be true after executing it. Prior work has used different mechanisms for specifying gadget types, including pattern matching on assembly instructions [21, 38] and expression tree matching [16]. We found postconditions to be more natural than these mechanisms. An instruction sequence $\mathcal{I}$ satisfies a postcondition $\mathcal{B}$ if and only if the post condition is true after running $\mathcal{I}$ from any starting state. The starting state consists of assignments to registers and memory. The full list of gadget types that Q can recognize is in Table 2, along with the corresponding semantic definition postconditions.

4.2.2 Semantic Analysis

Given an instruction sequence $\mathcal{I}$ and a semantic definition $\mathcal{B}$, Q must decide if $\mathcal{I}$ will satisfy $\mathcal{B}$. For this, we use a well-known technique from program verification for computing the *weakest precondition* of a program [15, 17, 23]. At a high level, the weakest precondition $\text{WP}(\mathcal{I}, \mathcal{B})$ for instructions $\mathcal{I}$ and postcondition $\mathcal{B}$ is a boolean precondition that describes when $\mathcal{I}$ will terminate in a state satisfying $\mathcal{B}$.

We use weakest preconditions in Q to verify whether the semantic definition of a gadget always holds after executing the instruction sequence $\mathcal{I}$. To do this, we check if

$$\text{WP}(\mathcal{I}, \mathcal{B}) \equiv true. \quad (1)$$

If this formula is valid, then $\mathcal{B}$ always holds after executing $\mathcal{I}$, and we can conclude that $\mathcal{I}$ is a gadget with the semantic type $\mathcal{B}$.

Our first prototype used only this semantic analysis. We found that it was too slow to be practical. We sped up the entire process by performing a number of random concrete executions, and evaluating each $\mathcal{B}$ concretely to see if it was true. If $\mathcal{B}$ was false for any concrete input, then the instruction sequence could not be a gadget for that gadget type. Thus, we only need to invoke the more expensive weakest precondition process when $\mathcal{B}$ is true for every random concrete execution.

Random concrete execution can also be used to infer possible parameter values (shown in Table 2) using dynamic analysis. For instance, by looking at the values of all registers, and the addresses that memory was read from, Q can compute a set of possible offsets for the LOADMEMG gadget type.

As an example of how a gadget type is tested, consider the LOADMEMG gadget type in Table 2. LOADMEMG gadgets operate on two registers: the output register and the address register. Each LOADMEMG gadget has two parameters that are specific to a particular instruction sequence $\mathcal{I}$. These will be found using dynamic analysis as described above. For instance, the instruction sequence `movl 0xc(%eax), %ebx; ret` is a LOADMEMG gadget with parameters {# Bytes $\leftarrow$ 4} and {Offset $\leftarrow$ 12} and registers {OutReg $\leftarrow$ %ebx} and {AddrReg $\leftarrow$ %eax}. The semantics for this instruction sequence would be %ebx $\leftarrow$ M[%eax + 12]. Q converts this to $\texttt{final}(\%ebx) = \texttt{initial}(M[\%eax + 12])$, which is the postcondition $\mathcal{B}$ that is checked for validity.

4.2.3 Gadget Discovery Algorithm

Our techniques for gadget discovery consist of two algorithms. The first, shown in Algorithm 1, tests whether or not the semantics of an instruction sequence $\mathcal{I}$ match those of any gadget type using randomized concrete testing and validity checking of the weakest precondition. Algorithm 1 also outputs some metadata (not shown) about each gadget for use in other Q algorithms, including the gadget's address, stack offset, and any registers that the gadget clobbers. The second algorithm iterates over the executable bytes of the source program, disassembles them, and calls the first algorithm as a subroutine. This is similar to the Galileo [41] algorithm, and so we do not replicate it here.

Algorithm 1 Automatically test an instruction sequence $\mathcal{I}$ for gadgets

```
Input: I, numRuns, gadgetTypes[]
for i = 1 to numRuns do
    outState[i] ← I(Random input)
end for
for gtype ∈ gadgetTypes do
    B ← postconditions[gtype]
    consistent ← true
    for j = 1 to numRuns do
        if B(outState[j]) ≡ false then
            consistent ← false
        end if
    end for
    if consistent = true then {Possibly a gadget of type gtype}
        F ← wp(I, B)
        if decisionProc(F ≡ true) = Valid then
            output {Output gadget I as type gtype}
        end if
    end if
end for
```

Name	Input	Parameters	Semantic Definition
NOOPG	—	—	Does not change memory or registers
JUMPG	AddrReg	Offset	**EIP** ← **AddrReg** + **Offset**
MOVEREGG	InReg, OutReg	—	**OutReg** ← **InReg**
LOADCONSTG	OutReg, Value	—	**OutReg** ← **Value**
ARITHMETICG	InReg1, InReg2, OutReg	$\Diamond_b$	**OutReg** ← **InReg1** $\Diamond_b$ **InReg2**
LOADMEMG	AddrReg, OutReg	# Bytes, Offset	**OutReg** ← M[**AddrReg** + **Offset**]
STOREMEMG	AddrReg, InReg	# Bytes, Offset	M[**AddrReg** + **Offset**] ← **InReg**
ARITHMETICLOADG	OutReg, AddrReg	# Bytes, Offset, $\Diamond_b$	**OutReg** $\Diamond_b$← M[**AddrReg** + **Offset**]
ARITHMETICSTOREG	InReg, AddrReg	# Bytes, Offset, $\Diamond_b$	M[**AddrReg** + **Offset**] $\Diamond_b$← **InReg**

Table 2: Types of gadgets that Q can find. M[addr] means accessing memory at address addr. $\Diamond_b$ means an arbitrary binary operation. **a** ← **b** denotes that final value of **a** equals the initial value of **b**. X $\Diamond_b$← Y is short for X ← X $\Diamond_b$ Y.

4.3 Gadget Arrangement

Q acts similar to a compiler — it reads in programs written in QooL (discussed below) and tries to implement them in terms of the gadgets shown in Table 2. The gadgets define an instruction set architecture. Thus, we can use some techniques from compiler theory. However, Q must deal with several hard problems not faced by most compilers:

- Only a few registers can be used for moving, accessing memory, and performing arithmetic operations.
- Most instructions will clobber (modify) the majority of available registers.
- Some instruction types may not be available at all.

Although we use existing compiler techniques when possible, many of the standard compiler techniques break down.

4.3.1 Q's Language: QooL

Users write the target program in Q's high level language, QooL, which is displayed in Table 3. QooL enables the user to easily interact with the exploited program's environment. For instance, the attacker can do this by calling a function (e.g., `system`), overwriting values in memory, or copying and running a binary payload (when W⊕X is not present or has been disabled by first calling `mprotect` or a similar function). QooL is not Turing-complete; we discuss this further in Section 8.

4.3.2 Arrangements

One of the essential tasks of a compiler is to perform *instruction selection*, since there are many combinations of instructions that can implement a given computation. The gadget architecture is no exception, as there are many ways of combining gadget types to produce a particular computation. We specify each combination of gadgets using a *gadget arrangement*.

```
<exp> ::=
    LoadMem <exp> <type>
    | BinOp <binop_type> <exp> <exp>
    | Const <int> <type>
<stmt> ::=
    StoreMem <exp> <exp> <type>
    | Assign <var> <exp>
    | CallExternal <func> <exp list>
    | Syscall
```

Table 3: Grammar for our high level language, QooL.

A gadget arrangement is a tree in which the vertices represent gadget types[5], and an edge labeled *type* from a to b means that the output of gadget a is used for the *type* input in gadget b. An example arrangement is shown in Figure 3.

One simple algorithm for performing instruction selection (or selecting a gadget arrangement, in our case) is the maximal munch algorithm [1]. Maximal munch assumes that any instruction selected as the best will always be available for use. This assumption makes sense in a traditional compiler, since on a normal architecture there are few restrictions on when instructions can be used.

A gadget arrangement algorithm cannot make such assumptions. Any particular gadget type chosen by maximal munch might not be available at that point in the program because Q did not find any or the registers in the gadgets are not compatible with other gadgets needed.

Instead of using maximal munch, Q employs *every munch*. Rather than selecting only one arrangement of

[5] Vertices also include parameters that are relevant to the computation, such as binary operator type and number of bytes for memory operations.

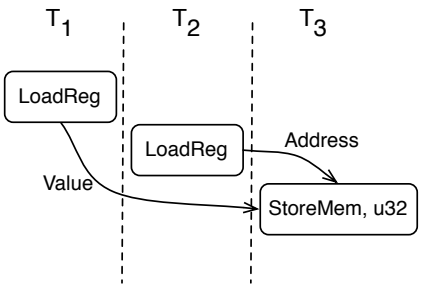

Figure 3: A gadget arrangement for storing a constant value to a constant address. A possible schedule for the arrangement is denoted by the time slots T_i's.

gadget types as maximal munch would, every munch lazily builds a tree representing all possible ways that gadget types can be arranged to perform a computation. This is done by recursively applying munch rules to the program being compiled.

4.3.3 Munch Rules

Each QooL language construct has at least one munch rule that can implement the construct in terms of the implementations of its subexpressions. For instance, the obvious munch rule for the `StoreMem` statement is to use a StoreMemG gadget, which we show below in ML-style pseudo code.

```
1 munch = function
2 | StoreMem(e1, e2, t) ->
3   let e1l = munch e1 in
4   let e2l = munch e2 in
5   (* For each e1g,e2g in Cartesian
6      product of e1l and e2l do: *)
7   add_output (StoreMemG(addr=e1g,
       value=e2g, typ=t));
```

Our initial implementation only contained these obvious rules. We quickly found that it could not find payloads for most binaries.

We found that, in practice, many binaries do not contain gadgets for directly storing to memory (StoreMemG in Table 2). We provide evidence of this in Section 7.1. However, if Q can learn or set the value in memory to 0 or -1, it can use an ArithmeticStoreG gadget with mathematical identities to write an arbitrary value. As one example, Q can write zero to memory by bitwise and'ing the memory location with zero, and then adding the desired number. The example below shows the complicated return oriented program Q discovered for writing a single byte to memory with bitwise or, using gadgets from `apt-get`. More straightforward options were not available.

```
; Load eax: -1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebx: address-0x5e5b3cc4
pop %ebx; pop %ebp; ret
; Write -1
or %al, 0x5e5b3cc4(%ebx); pop %edi;
   pop %ebp; ret
; Load eax: value + 1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebp: address-0xf3774ff
pop %ebp; ret
; Add value + 1
add %al,0xf3774ff(%ebp);
   movl $0x85, %dh; ret
```

4.4 Gadget Assignment

Q must determine if a gadget arrangement can be satisfied using the gadgets it discovered in the source program. This process is called gadget assignment. The goal is to assign gadgets found during discovery to the vertices of arrangements, and see if the assignment is compatible. After a successful gadget assignment, the output is a mapping from gadget arrangement vertices to concrete gadgets. It is straightforward to print a ROP payload with this mapping.

Gadget assignments need a schedule, since the gadgets must execute in some order. Selecting a valid schedule is not always easy because there are data dependencies between different gadgets. For instance, if the gadget at T_2 clobbers (overwrites) the Value register in Figure 3, the gadget at T_3 will not receive the correct input. To resolve such dependencies between gadgets, a gadget assignment and corresponding schedule must satisfy these properties:

Matching Registers Whenever the result of gadget a is used as input $type$ to gadget b, then the two registers should match, i.e., $\mathit{OutReg}(a) = \mathit{InReg}(b, type)$.

No Register Clobbering If the output of gadget a is used by gadget b, then a's output register should not be clobbered by any gadget scheduled between a and b. For example, for the schedule shown in Figure 3, the LoadConstG operation during T_2 should not clobber the result of the previous LoadConstG that happened during T_1.

We say that a gadget assignment and schedule are compatible when the above properties hold, and that a gadget arrangement that has a compatible assignment and schedule is satisfiable.

Although deciding whether a given gadget schedule and assignment are compatible is straightforward (i.e.,

just ensure the above properties are satisfied), creating a practical algorithm to search for satisfiable arrangements is more complicated. The most straightforward approach is to iterate over all possible arrangements, schedules, and assignments, but this is simply too inefficient.

Instead, our key observation is that if a gadget arrangement **GA** is unsatisfiable, then any **GA'** that contains **GA** as a subtree is unsatisfiable as well. Our algorithm attempts to satisfy iteratively larger subtrees until it fails, or has satisfied the entire arrangement. If the algorithm fails on a subtree, it aborts the entire arrangement. Since most arrangements are unsatisfiable, this saves considerable time. (If most arrangements are satisfiable, the search will not take very long anyway.)

Our assignment algorithms are found in Algorithms 2 and 3. Algorithm 2 is a naive search over a schedule for all possible gadget assignments. Algorithm 3 is a caching wrapper that caches results and calls Algorithm 2 on iteratively larger subtrees. It stops as soon as it finds a subtree which cannot be satisfied. Q calls Algorithm 3 on each possible gadget arrangement until one is satisfiable or there are none left.

The algorithms make use of several data structures:

- **C**: $\mathcal{V} \rightarrow \{0, 1, ?\}$ is a cache that maps a gadget arrangement vertex to one of true, false, or unknown.
- **S**: $\mathcal{V} \rightarrow \mathcal{N}$ represents the current schedule as a one-to-one mapping between each vertex and its position in the schedule.
- **G**: $\mathcal{V} \rightarrow \mathcal{G}$ is the current assignment of each vertex to its assigned gadget.

Q can also search for assignments that meet other constraints. For instance, Q can search for assignments that would result in a payload smaller than a user-specified size. This is useful because ROP payloads are typically larger than conventional payloads, and vulnerabilities usually limit the number of payload bytes that can be written.

Algorithm 2 Find a satisfying schedule and gadget assignment for **GA**

```
Input: S, G, nodeNum
V ← S^{-1}(nodeNum) {Obtain vertex in GA for nodeNum}
if V = ⊥ then {Base case to end recursion}
    return true
end if
gadgets ← GadgetsOfType(GadgetType(V))
for all g ∈ gadgets do
    if IsCompatible(G, nodeNum, g) then {Ensure g is compatible with all gadgets before time slot nodeNum}
        if Algorithm 2(S, G[V ← g], nodeNum + 1) then {Try to schedule later schedule slots}
            return true
        end if
    end if
end for
return false {No gadgets matched}
```

Algorithm 3 Iteratively try to satisfy larger subtrees of a **GA**, caching results over all arrangements.

```
Input: GA, C
for all GA' ∈ Subtrees(GA) do {In order from shortest to tallest}
    if C(GA') = ? then
        C(GA') ← exists S ∈ Schedules(GA') such that Algorithm 2(S, Empty, 0) = true
    end if
    if C(GA') = false then {Stop early if a subtree cannot be satisfied}
        return false
    end if
end for
return C(GA) {Return the final value from the cache}
```

5 Creating Exploits that Bypass ASLR and W⊕X

In the previous section, we described how to generate return oriented *payloads*. If an attacker can redirect execution to the payload in the memory space of the vulnerable program by creating an exploit, then the computation specified by the payload will occur. In this section, we explain how Q can automatically create such an exploit when given an input exploit that does not bypass ASLR and W⊕X.

We call this the exploit hardening problem. Specifically, in the exploit hardening problem we are given a program $\mathcal{P}$ and an input exploit that triggers a vulnerability. The input exploit can be an exploit that does not bypass defenses, or can even be a proof of concept crashing input. The goal is to output an exploit for $\mathcal{P}$ that bypasses W⊕X and ASLR.

Intuitively, the input exploit should provide useful information about a vulnerability in $\mathcal{P}$. Q uses this information to consider other inputs that follow the execution path of the input exploit (i.e., the sequence of conditional branches and jumps taken by an execution of the input) on $\mathcal{P}$, and attempts to find a new input that uses a return-oriented payload instead (Section 4).

Q does not always succeed (e.g., sometimes it returns with no exploit), but we show that it works for real Linux and Windows vulnerabilities in Section 7. The fact that our system works with even a few real exploits means that an attacker can sometimes download an exploit and automatically harden it to one that works even when W⊕X and ASLR are enabled.

5.1 Background: Generating Formulas from a Concrete Run

There can be a very large number of inputs along the vulnerable path. Rather than trying to reason about each input individually, we build a logical constraint formula representing all inputs that follow the vulnerable path. Such constraint formulas have been used in many research areas, including automatic test case generation, automatic signature creation, and others [5, 7, 23, 40].

Generating constraint formulas from an input involves two steps. First, we record at the binary level the concrete execution of the vulnerable program running on the input exploit; we call such a recording an execution trace. Our recording tool incorporates dynamic taint analysis [11, 33, 40] to keep track of which instructions deal with tainted (or input-derived) data. Our tool uses this information to 1) record only the instructions that access or modify tainted data, for performance reasons; and 2) halt the recording once control-hijacking takes place (i.e., when the instruction pointer becomes tainted).

After recording the concrete execution, Q symbolically executes [7, 40] the target program, following the same path as in the recording. Symbolic execution is similar to normal execution, except each input byte is replaced with a symbol (e.g., s_i for input byte i). Any computation involving a symbolic input is replaced with a symbolic expression. Computations not involving a symbolic input are computed as normal (i.e., using the processor). Any constraints on the inputs to ensure that execution would be guided down the same path as the execution trace are stored in the constraint formula Π.

Before performing any analysis, we use the Binary Analysis Platform [3] to raise binary code into an intermediate language that is better suited to program analysis. This frees our analysis from needing to understand the semantics of each assembly instruction.

5.2 Exploit Constraint Generation

The constraint formula Π describes all inputs that follow the vulnerable path. In this paper, we are only interested in inputs that hijack control to our desired computation. We build two constraints, α (control flow) and Σ (computation), that exclude any inputs that do not work as exploits. α maps to true only if a program's control flow has been diverted, and Σ maps to true only if the payload for some desired computation is in the exploit.

5.2.1 Assuring Control Flow Hijacking

α takes the form `jumpExp` $=$ `targetExp`, where `jumpExp` is the symbolic expression representing the target of the jump that tainted the instruction pointer, and `targetExp` depends on the type of exploit.

The value of `jumpExp` can be obtained from the execution trace. Since the trace halts when the program jumps to a user-derived address, `jumpExp` is simply the symbolic expression for the target of this jump. Consider the following program.

```
1  x := 2*get_input()
2  goto x
```

Our trace system would halt the above program at Line 2, because the program jumps to a user-derived address. The symbolic jump expression from symbolic execution of the program is $2 * s_1$. α for this program would be $2 * s_1 =$ `targetExp`.

For a typical stack exploit, `targetExp` $= \&(shellcode)$, where & means the address of. With a return oriented payload, this would usually be `targetExp` $= \&($`ret`$)$. This assumes that the ROP payload is located in memory at the address in %esp. If not, Q can use a pivot, which its ROP system can automatically find. For instance, `targetExp` $= \&($`xchg %eax, %esp; ret`$)$ would transfer control to the ROP payload pointed to by %eax.

5.2.2 Assuring Computation

Computation constraints ensure that the computation payload is available in memory at the proper address at the time of exploitation. For instance, computation constraints for a `strcpy` buffer overflow would be unsatisfiable for a payload containing a null byte, since this would result in only part of the payload being copied.

Computation constraints take the form $\Sigma = (mem[$`payloadBase`$] =$ `payload`$[0] \wedge \ldots \wedge mem[$`payloadBase`$ + n] =$ `payload`$[n])$, where `payloadBase` denotes the starting address of the payload in memory, and `payload` denotes the bytes in the payload (e.g., the ROP payload from Section 4). When using a basic ROP payload, `payloadBase` will be set to %esp, since that is where a `ret` will start executing. When using a pivot, this value will depend on the pivot in the natural way.

5.2.3 Finding an Exploit

By combining these constraints with Π, which only holds for inputs following the vulnerable path, we can create a

constraint formula that only describes exploits along the vulnerable path:

$$\Pi \wedge \alpha \wedge \Sigma. \quad (2)$$

Any assignment to the initial program state that satisfies this constraint formula is an exploit for the program semantics recorded in the trace. We use an off the shelf decision procedure, `STP` [19], to solve the formulas.

6 Implementation

The ROP component (Section 4) of Q is built on top of the BAP framework [3]. The implementation for the gadget discovery, arrangement, and assignment phases comprises 4,585 lines of ML code. The ROP system uses the `STP` [19] decision procedure to determine the validity of generated weakest preconditions.

Q's exploit hardening component (Section 5) itself consists of a tracing (recording) component and an analysis component. We implemented the tracing tool using the Pin [28] framework, which allows analysis code to instrument a running process and take measurements in between instruction execution. Our tool is optimized to only record instructions that are considered to be user-derived; the user can mark any input coming from files, network sockets, environment variables, or program arguments as being user-derived, and can record processes that fork (e.g., network daemons). The tracing component is written in C++, and includes 2,102 lines of code written for this project.

The analysis portion of the hardening system is implemented in the BAP [3] framework. It consists of components that 1) lift the recorded assembly instructions into the BAP intermediate language, 2) symbolically execute the trace, obtaining the constraint formula Π, and 3) compute the constraints α and Σ. Our analysis tool then uses `STP` [19] to find a satisfying answer to the resulting constraint formula, and uses the result to build an exploit. It also fully understands Windows SEH (structured exception handler) exploits, in which the exception handler is overwritten. The analysis implementation is written in ML, and includes 1,090 new lines of code for this project.

All components of Q are fully capable of reasoning about Windows and Linux binaries.

7 Evaluation

We evaluate Q's capabilities to produce ROP payloads and harden exploits in this section.

7.1 Return Oriented Programming

Applicability We would like to know how often Q can build ROP payloads when given a random source program P. To evaluate this, we ran Q on all of the 1,298 ELF programs in `/usr/bin` on an author's Ubuntu 9.10 desktop machine and tried to generate various return oriented payloads. We then discarded the results for the 66 programs that were marked as ASLR-compatible (PIE). We used Linux programs for our corpus because it is easier to gather a typical set of Linux programs than for Windows. For each program P, we consider if Q can create a ROP payload to:

Call functions also called by P External functions called by P have an entry in the program's Procedure Linkage Table (PLT). Q calls the PLT entries directly; if the external function has not been loaded, the dynamic loader will be invoked to load it before transferring control to the called function.

Call external functions in libc Calling external functions that do not have a PLT entry is more complicated. For this, we build on a technique for calculating the address of functions in libc even when libc is randomized [39]. This involves more computation than the above case, and so is more likely to be unsatisfiable.

Write to memory We consider a payload that writes four bytes to an arbitrary address.

For each of the target programs above, we measure whether our system can create a payload for it using instruction sequences taken from each source program in our corpus. We consider an attempt successful if our system successfully builds a payload. Note that the attacker must still find a way to load the payload into memory and redirect control to it for it to be used as an exploit.

The results of this experiment are shown in Figure 4. The probability of success for the above payload types is plotted as a function of source program size. The Call/-Store line represents the *Call functions also called by P* and *Write to memory* cases above, since the results are visually indistinguishable. The Call (libc) line represents *Call external functions in libc*.

The results support the claim that ROP is more difficult when there is less binary code. Even so, Q is able to call linked functions and store arbitrary memory bytes to arbitrary locations in 80% of binaries that are at least 20KB. Q can also call any function in libc in 80% of binaries 100KB or larger[6].

[6]The fact that Q generated payloads for so many binaries was disturbing to the author whose machine the programs came from.

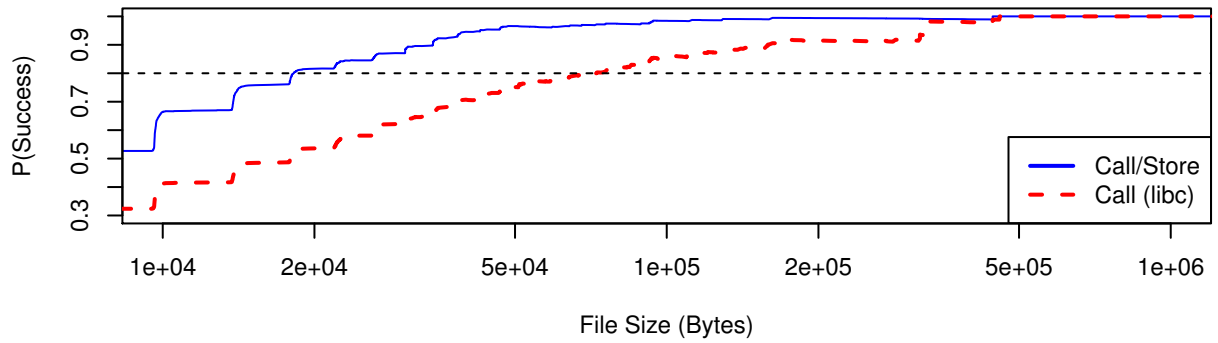

Figure 4: The probability that Q can generate various payload types, shown as a function of source file size. As expected, the probability grows with file size. The percentage is calculated over non position independent executables. Q can call linked functions in 80% of programs that are 20KB or larger, and can call any function in linked shared libraries in 80% of programs that are at least 100KB in size.

Efficiency While we found that semantic gadget discovery techniques are useful for finding gadgets, they are not very fast. In our implementation, we found that adding a concrete randomized testing stage increased Q's performance. To measure this, we collected a random sample of 32 programs from our `/usr/bin` dataset and ran gadget discovery. For each program, we ran Q twice, once with randomized testing enabled, and once disabled. Figure 5 shows a boxplot of the elapsed wall times when running with 16 active threads. (The time difference would be greater with fewer threads, but the experiment would take a very long time to complete for the non-randomized cases.) As expected, Q runs faster when randomized testing is enabled.

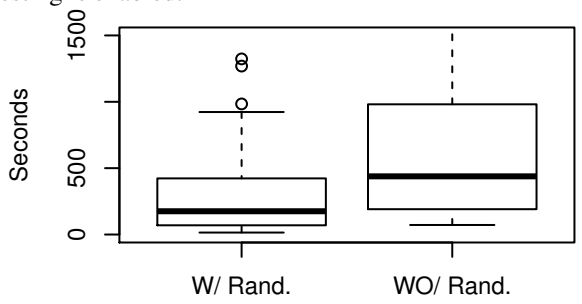

Figure 5: Boxplots of the time it takes to discover gadgets from a program for a random sample of 32 programs, when randomized testing is enabled and disabled.

Sizes Our results from Figure 4 show that larger programs are generally easier to build return oriented programs from. Figure 6 shows the sizes of the programs in our experiments, and compares them to the binaries used in prior research, libc [38], the iPhone library [16], and the windows kernel [21]. We note that these binaries are significantly larger than most `/usr/bin` programs.

Gadget Frequency Figure 7 shows the frequency of various types of gadgets in programs larger than 20KB.[7] It offers some insight on why ROP on small binaries is difficult. The most useful gadget types, like StoreMemG and LoadMemG, are not very common. Instead, combined gadgets like ArithmeticStoreG are more prevalent. This is not surprising, given that compilers try to combine operations to optimize efficiency. These results are what inspired Q's gadget arrangement system, which can cope with missing gadget types.

7.2 Exploit Hardening

To evaluate exploit hardening, we tested it with a variety of publicly available exploits for Linux and Windows. We consider each experiment a success if Q can harden a public exploit for real software by producing working exploits that bypass W⊕X and ASLR. We do not expect that our system will always produce a hardened exploit.

We compiled each vulnerable program from source when possible, disabled all defenses (including ASLR and W⊕X), and then verified that the exploit at least crashed the vulnerable program. We then ran the exploit through the exploit hardening component of Q, and created two payloads that bypass W⊕X and ASLR. These payloads 1) call a linked function and 2) call `system(``w'')`

[7] These results are after a pre-processing step that throws away redundant gadgets. A gadget g_1 is redundant to g_2 if they both have the same type and input registers, and g_1 clobbers a superset of the registers that g_2 clobbers. This is why there is only one NoOpG gadget type listed for all programs, even though every `ret` instruction can be used as a NoOpG.

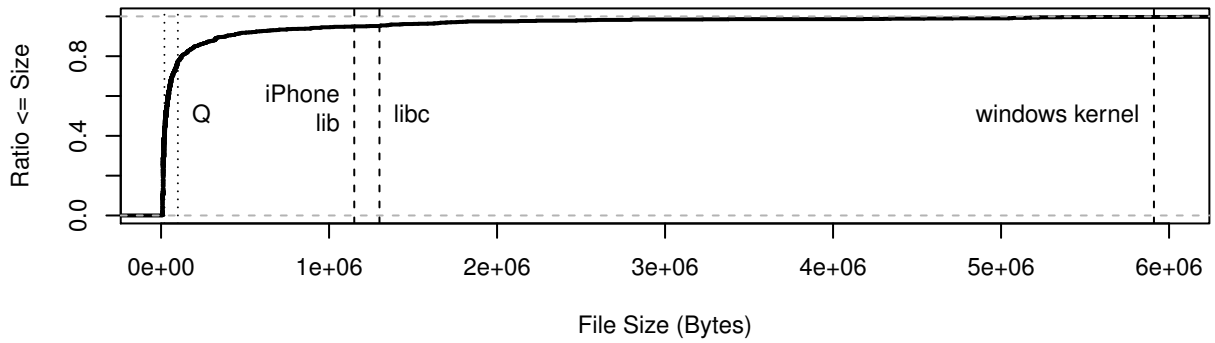

Figure 6: The empirical cumulative distribution function of the file sizes in `/usr/bin`. In this graph, a point at (x, y) means that $100y$ percent of the files in `/usr/bin` have a size less than or equal to x bytes. We also show the sizes of the iPhone libsystem library [16], libc [38] and the windows kernel [21], which prior work has targeted. libc and the iPhone library are both larger than 95% of the programs in our corpus, while the windows kernel is larger than 99%. We plot dotted lines at 20 and 100KB for reference; these are the sizes at which Q works well, as shown in Figure 4.

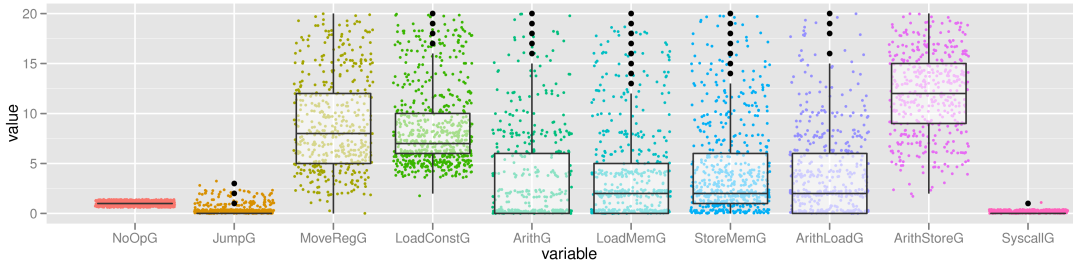

Figure 7: The frequency of various types of gadgets in `/usr/bin` programs larger than 20KB.

on Linux or `WinExec(``calc.exe'')` on Windows. We tested these two exploits with ASLR and W⊕X enabled. The results of these experiments are shown in Table 4.

We found that our system was able to harden exploits for several large, real programs. In general, our system performed as expected: it only output exploits that worked, and in some cases reported it could not produce a hardened exploit.

8 Discussion

Ret-less ROP When we designed Q, no one had shown that ROP was possible without using `ret`-like instructions. Since then, Checkoway, et al. have shown [8] that it is possible to create a Turing-complete gadget set that does not use `ret` instructions. Their gadgets have control flow preservation preconditions. For example, the gadget `pop %eax; jmp *%edx` only preserves control flow if %edx is preset to the next gadget address. Q does not make any assumptions about the preconditions for a gadget when considering control flow preservation, which prevents it from finding gadgets of the above form. We leave it as future work to determine whether it is possible to automatically construct ROP exploit payloads that do not use `ret` instructions.

Side effects Q conservatively handles side effects by discarding any instruction sequence that might cause the program to crash, such as a pointer dereference. As one example, `pushl %eax; popl %ebx; ret` will move the value in %eax to %ebx. Since a MOVEREGG gadget does not intentionally use memory, however, Q would discard this gadget. We plan to add a more advanced memory analysis that can statically detect when a memory access will be safe, which will allow Q to use more gadgets.

Turing completeness Q's language for describing target programs, QooL, is not Turing-complete. Our early tests revealed that the ARITHMETICG gadgets needed for conditional jumps, such as equality tests, were often unavailable in small programs. As a result, we focused on the gadgets needed for practical exploitation, rather than

Program	Reference	Tracing	Analysis	Call Linked	Call System	OS	SEH
Free CD to MP3 Converter	OSVDB-69116	89s	41s	Yes	Yes	Win	No
FatPlayer	CVE-2009-4962	90s	43s	Yes	Yes	Win	Yes
A-PDF Converter	OSVDB-67241	238s	140s	Yes	Yes	Win	No
A-PDF Converter	OSVDB-68132	215s	142s	Yes	Yes	Win	Yes
MP3 CD Converter Pro	OSVDB-69951	103s	55s	Yes	Yes	Win	Yes
rsync	CVE-2004-2093	60s	5s	Yes	Yes	Lin	NA
opendchub	CVE-2010-1147	195s	30s	Yes	No	Lin	NA
gv	CVE-2004-1717	113s	124s	Yes	Yes	Lin	NA
proftpd	CVE-2006-6563	30s	10s	Yes	Yes	Lin	NA

Table 4: A list of public exploits hardened by Q. For each exploit, we record how long the trace and analysis components took to run, and report if Q produced hardened exploits that call 1) a linked function, and 2) `system` or `WinExec`.

striving for Turing-completeness.

9 Related Work

Return Oriented Programming Krahmer was the first to propose using borrowed code chunks [25] from the program text to perform meaningful actions. Later, Shacham showed in his seminal paper [41] on ROP that a set of Turing complete gadgets can be created using the program text of libc. Shacham developed an algorithm that put instruction sequences into trie form to help a human manually select useful instruction sequences.

Since then, several researchers have investigated how to more fully automate ROP [16, 21, 38]. Dullien and Kornau [16, 24] automatically found gadgets in mobile support libraries (on order of 1,000KB), and Roemer [38] demonstrated it was possible to automatically discover gadgets in libc (1,300KB). Hund [21] used gadgets from `ntoskrnl.exe` (3,700KB) and `win32k.sys` (2,200KB). In contrast, our techniques often only have 20KB of binary code to create gadgets from, because generally only small code modules are unrandomized in user-mode exploitation contexts. Previous work focusing on such small code bases was mostly or entirely manual; for instance, Checkoway, et al. manually crafted a Turing complete set of gadgets from 16KB of Z80 BIOS [9].

Automatic Exploitation Our exploit hardening system (Section 5) is related to existing automatic exploitation research [2, 5, 20, 26]. In automatic exploitation, the goal is to automatically find an exploit for a bug when given some starting information (such as a patch [5], guiding input [20, 26], or program precondition [2]). Some automatic exploitation research focuses on creating an input that triggers a particular vulnerability [5, 18, 26], but does not focus on control flow exploitation, which is one of the focuses of our work. Our techniques can use the inputs produced by these projects as an input exploit, and harden them so that they bypass W⊕X and ASLR.

We are only aware of one other project that considers creating an exploit given another exploit [20]; in this case the input exploit only causes a crash. Our work uses symbolic execution to reason about other inputs that take the same path as the input exploit. In contrast, Heelan [20] tracks data dependencies between the desired payload bytes and the input bytes, but does not ensure that control flow will stay the same and preserve the observed data dependencies. As a result, his approach is heuristic in nature, but is likely to be faster than ours.

Related Attacks Other researchers have previously used simple ROP gadgets in the `.text` section of binaries to calculate the address of functions in libc [39]. Unfortunately, this is insufficient to make arbitrary function calls when ASLR is enabled, because many functions require pointers to data. Recall from Section 2 that all modern operating systems except for Mac OS X randomize the stack and heap, thus making it difficult for an attacker to introduce argument data and know a pointer to its address. QooL (Section 4.3.1) allows target programs to write payloads to known addresses, typically in the `.data` segment, which eliminates this problem.

A recent attack developed concurrently with Q [27] can also write data to known constant memory locations, and thus can also make arbitrary function calls in the W⊕X and ASLR setting. This attack uses repeated `strcpy` return-to-libc calls to copy data from the binary itself to a specified location. In contrast, our attack uses ROP gadgets discovered by Q.

There are specialized attacks against W⊕X and ASLR

that are only applicable inside of a browser, such as JIT spraying [4, 43]. The downside is that they are not applicable to all programs.

Related Defenses The most natural way of defeating ROP is to randomize all executable code. For instance, we are not able to deterministically attack position independent executables in Linux, because we do not know where any instruction sequences will be in memory. Operating systems have chosen not to randomize all code in the past because of performance and compatibility issues; these reasons should now be reevaluated considering the new evidence that allowing even small amounts of unrandomized code can enable an attacker to use ROP payloads.

Other defenses against ROP exist. One defense is to dynamically instrument running programs and look for sequences of instructions that contain returns with few instructions spaced between [10, 12]. The assumption is that normal code will generally execute non-trivial amounts of code in between `ret` instructions, whereas ROP code will not.

A similar defense is to ensure that the call chain of a program respects the stack semantics, i.e., that a `ret` will only transfer control to a program location that previously executed a `call` instruction. Such techniques [13, 37] are implemented using a shadow stack that is maintained outside of normal memory space. Both of these defenses make the assumption that ROP must be performed using the `ret` instruction.

Unfortunately for defenders, researchers [8] have recently shown that it is possible to perform ROP on x86 without using `ret` instructions at all, which is enough to bypass these schemes without modifications. However, the proof of concept techniques required access to large libraries, which are randomized in modern operating systems. It remains an open question whether such attacks are possible in modern user-mode exploitation contexts, when little unrandomized code is available.

10 Conclusion

We developed return oriented programming (ROP) techniques that work on small, unrandomized code bases as found in modern systems. We demonstrated that it is possible to synthesize ROP payloads for 80% of programs larger than 20KB, implying that even a small amount of unrandomized code is harmful. We also built an end-to-end exploit hardening system, Q, that reads as input an exploit that does not bypass defenses, and automatically hardens it to one that bypasses ASLR and W⊕X. Our techniques and experiments demonstrate that current ASLR and W⊕X implementations, which allow small amounts of code to be unrandomized, continue to allow ROP attacks. Operating system designers should weigh the dangers of such attacks against the performance and compatibility penalties imposed by randomizing all code by default.

References

[1] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[3] Binary Analysis Platform (BAP). `http://bap.ece.cmu.edu`.

[4] D. Blazakis. Interpreter exploitation. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010.

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 27–38, 2008.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

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

[9] S. Checkoway, J. A. Halderman, U. Michigan, A. J. Feldman, E. W. Felten, B. Kantor, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of the Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, Aug. 2009.

[10] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the Information Systems Security Conference*, 2009.

[11] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[12] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2009.

[13] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender:

a detection tool to defend against return-oriented programming attacks. In *Proceedings of the ACM Symposium on Information, Computer, and Communication Security*, 2011.

[14] Debian Developers. Debian hardening. http://wiki.debian.org/Hardening?action=recall&rev=34. Accessed: August 8th, 2010.

[15] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[16] T. Dullien and T. Kornau. A framework for automated architecture-independent gadget search. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Aug. 2010.

[17] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[18] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of api-level exploits. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2005.

[19] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the Conference on Computer Aided Verification*, pages 524–536, July 2007.

[20] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

[21] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the USENIX Security Symposium*, 2009.

[22] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual – volume 3A: System programming guide, part 1. Document number 253668, 2010.

[23] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, Feb. 2010.

[24] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universität Bochum, 2009.

[25] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf, 2005.

[26] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *International Conference on Dependable Systems and Networks*, 2008.

[27] L. D. Long. Payload already inside: data re-use for ROP exploits. Technical report, Blackhat, 2010.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2005.

[29] Microsoft Software Developer Network. Windows vista ISV security. http://msdn.microsoft.com/en-us/library/bb430720.aspx.

[30] Microsoft Support. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. http://support.microsoft.com/kb/875352/EN-US/.

[31] I. Molnar. exec-shield linux patch. http://people.redhat.com/mingo/exec-shield/.

[32] T. Müller. ASLR smack & laugh reference. Technical report, RWTH-Aachen University, 2008. http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[34] PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[35] PaX Team. Pax non-executable stack (nx). http://pax.grsecurity.net/docs/noexec.txt.

[36] A. R. Pop. DEP/ASLR implementation progress in popular third-party windows applications. http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf, 2010. Secunia.

[37] M. Prasad and T. cker Chiueh. A binary rewriting defense again stack-based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[38] R. G. Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009.

[39] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the Annual Computer Security Applications Conference*, pages 60–69, 2009.

[40] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

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

[42] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[43] A. Sotirov and M. Dowd. Bypassing browser memory protections. Technical report, Blackhat, 2008. http://taossa.com/archive/bh08sotirovdowd.pdf.

[44] Ubuntu Developers. Ubuntu security/features. https://wiki.ubuntu.com/Security/Features?action=recall&rev=52. Accessed: August 8th, 2010.