Removing ROP Gadgets from OpenBSD

Todd Mortimer
mortimer@openbsd.org

Abstract

Return Oriented Programming (ROP) is a common exploitation technique that reuses existing code fragments (gadgets) to construct shellcode in a compromised program. Recent changes in OpenBSD's compiler have started to reduce the number of gadgets in x86 and arm64 binaries, with the aim of making ROP exploitation more difficult or impossible. This paper will cover how ROP gadgets emerge from legitimate code, how OpenBSD's compiler removes these gadgets, and the effects on performance, code size, and ROP tool capabilities. We find that it is possible to meaningfully reduce the number of ROP gadgets in programs, and to effectively hinder ROP tool capabilities.

1 Background

Return oriented programming (ROP) [5] is an exploitation technique that uses fragments of existing programs in unintended ways to effect control over a compromised process. In contrast to traditional shellcode injection, ROP attacks inject a series of return addresses - a ROP Chain - into memory and which, when execution returns to the first address in the chain, cause execution to iterate through a series of small code fragments which have the same effect as traditional shellcode. ROP is a powerful technique in environments which disable simultaneous writable and executable memory (W$\oplus$X), since it does not rely on injecting executable code into program memory, but instead relies only on program fragments that already exist. These program fragments are called *gadgets*, and each gadget consists of a (typically small) sequence of instructions followed by a return. On aligned architectures, these returns are part of the intended instruction stream that make up the program, but on unaligned architectures such as x86, these returns can also emerge from jumping into the instruction stream at unintended offsets and causing the existing code to be interpreted differently from what was intended. ROP techniques have been used in attacks on real world systems, including recent attacks exploiting CVE-2018-5767[1], CVE-2018-7445[2] and CVE-2018-16865/6[3].

Numerous techniques have been proposed to mitigate against ROP exploits, including return address verification techniques [2] and control flow verification [1] which aim to prevent control flow being redirected towards a ROP chain. Attempts have also been made to attempt to remove or render unusable ROP gadgets themselves [4]. This paper describes ROP exploit mitigations in OpenBSD which are motivated by gadget reduction and removal, though some mitigations also verify return control flow through return address verification.

In order to mount a successful ROP attack against a vulnerable binary, the attacker must first catalogue all of the gadgets available in a given binary, then identify a sequence of gadgets which will result in their desired effect. This process of scanning binaries for gadgets and then constructing ROP chains which have a desired outcome is somewhat tedious and error prone, so numerous tools exist to make this easy, such as ROPGadget[4], ropper[5], angrop[6], or pwntools[7]. In this paper we will rely on the output from one of these tools, ROPGadget, to measure our effectiveness. Specifically, we will use the number of unique gadgets found by this tool to measure the effectiveness of gadget removal in the OpenBSD kernel and libc, which we have chosen because they are large and diverse binary objects, and are popular exploitation targets. ROPGadget also includes an option to generate a ROP chain that results in an exploited program executing a command shell. Obtaining a command shell is a common exploitation goal, since once an attacker has a command shell they can execute arbitrary other commands on the compromised system. We will use this feature to estimate the effectiveness of our efforts to impede mounting successful ROP attacks

[1] https://www.fidusinfosec.com/remote-code-execution-cve-2018-5767/
[2] https://www.coresecurity.com/advisories/mikrotik-routeros-smb-buffer-overflow
[3] https://www.openwall.com/lists/oss-security/2019/01/09/3
[4] https://github.com/JonathanSalwan/ROPgadget
[5] https://github.com/sashs/ropper
[6] https://github.com/salls/angrop
[7] http://docs.pwntools.com/en/stable/

against OpenBSD binaries. The output from the ROPGadget ropchain option is shown in Figure 1, and illustrates several important concepts in ROP attacks. First, the program scans the given binary and identifies all unique gadgets present - this output is shown first. Next, in order to successfully take control of the program and spawn a command shell, gadgets of certain *classes* must be found. These gadgets are listed after each [+] symbol, and show the address of the gadget and the instructions that will be executed when the program jumps to that address. The first class of gadget is a *write-what-where* gadget, which allows values to be moved between registers and memory locations. Next, a gadget must be found which can set the syscall number required for the *exec* system call - notice that these gadgets manipulate the value of the RAX register. The third class of gadget needed sets the arguments to the exec syscall, and these gadgets manipulate the RDI and RSI registers to set the arguments to the exec system call to be */bin/sh*. Finally, the last gadget type is the syscall gadget, which will execute the system call set up by the other gadgets. After identifying suitable gadgets in each class, ROPGadget will construct a ROP chain that will direct program flow through these gadgets in such a way to cause the program to execute a command shell. ROPGadget outputs the ROP chain as a python program which can easily be inserted into whatever exploit tool is being developed to target a specific program or binary. This level of ease and accessibility is typical of ROP tooling, and illustrates the ease with which ROP attacks can be mounted once a suitable vulnerability is identified which takes control of program execution.

We can see from this output that a variety of gadgets are required in order to mount successful ROP attacks on binaries, and that there are a large number of unique gadgets available in a typical binary. These observations motivate our approach of reducing the number of gadgets available in a typical binary - if we can reduce the number of unique gadgets enough then the remaining gadgets will be insufficient to mount a successful attack. In particular, if we can remove all gadgets of a particular class, then it may become impossible to mount some kinds of attacks against OpenBSD binaries, such as the *exec("/bin/sh")* attack shown in Figure 1. We therefore do not need to reduce the number of gadgets in a binary to zero in order to foil ROP attacks, we only need to remove enough gadgets, or enough types of gadgets, so that an attacker cannot cobble together a viable ROP chain.

2 Removing Gadgets

ROP gadgets depend on a sequence of instructions terminating on a return instruction. On aligned architectures, such as arm64, these return instructions are part of the intended instruction stream and are part of usual function epilogues. On unaligned architectures, such as x86/amd64, there are additional return instructions which arise when jumping into the instruction stream at offsets other than those corresponding to the intended stream of instructions. These *polymorphic gadgets* terminate on return instructions that are intended to be part of constants, multibyte instructions, or other artifacts in programs other than real return instructions. In this section, we discuss each kind of gadget and our techniques to remove or reduce them in compiled binaries.

2.1 Aligned Gadgets

Aligned gadgets terminate on intended return instructions as part of normal function epilogues. On aligned architectures these gadgets comprise some or all of the usual process of restoring register state before a function returns. On unaligned architectures these gadgets can also have entirely different effects depending on the offset where the gadget begins in the instruction stream. Examples of aligned gadgets on amd64 and arm64 are shown in Figures 2 and 3. Both of these examples are found in function epilogues, and we show both the bytes that make up the instruction and the instruction disassembly. Throughout this paper we will show both the bytes in the compiled binary and the disassembled instructions, since the interpretation of the bytes making up a compiled program is central to the concept of return oriented programming. For readers unaccustomed to inspecting program disassembly, it may be fruitful to use *objdump(1)* to disassemble and inspect various programs and libraries on their systems.

Figure 2: Aligned gadget on amd64

Bytes	Disassembly
`0f b6 c0`	`movzbl %al, %eax`
`5d`	`popq %rbp`
`c3`	`retq`

Figure 3: Aligned gadget on arm64

Bytes	Disassembly
`fe 03 05 aa`	`mov x30, x5`
`c0 03 5f d6`	`ret`

Since aligned gadgets terminate on function return instructions which are required for correct program operation, our strategy to prevent these return instructions being used in ROP gadgets will be to make them difficult to use outside of normal program flow. To this end, we insert interrupt instructions before the returns and then add instrumentation to the function that will allow normal program flow to jump over the interrupts. Program flow that starts at an offset other than the normal function entry point will fail to jump over the interrupts and abort.

Figure 1: ROPGadget ropchain against OpenBSD 6.3 libc

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.3/libc.so.92.3

Unique gadgets found: 8453
ROP chain generation
===========================================================
- Step 1 -- Write-what-where gadgets
        [+] Gadget found: 0x1f532 mov qword ptr [rsi], rax ; pop rbp ; ret
        [+] Gadget found: 0x3b62e pop rax ; ret
- Step 2 -- Init syscall number gadgets
        [+] Gadget found: 0xfa0 xor rax, rax ; ret
        [+] Gadget found: 0x38fe inc rax ; ret
- Step 3 -- Init syscall arguments gadgets
        [+] Gadget found: 0x4cd pop rdi ; pop rbp ; ret
        [+] Gadget found: 0x905ee pop rsi ; ret
- Step 4 -- Syscall gadget
        [+] Gadget found: 0x9c8 syscall
- Step 5 -- Build the ROP chain
        p = ''
        p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
        p += pack('<Q', 0x00000000002cd000) # @ .data
        p += pack('<Q', 0x000000000003b62e) # pop rax ; ret
        p += '/bin//sh'
        p += pack('<Q', 0x000000000001f532) # mov qword ptr [rsi], rax ; pop rbp ; ret
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
        [elided ...]
        p += pack('<Q', 0x00000000000038fe) # inc rax ; ret
        p += pack('<Q', 0x00000000000009c8) # syscall
```

RETGUARD

RETGUARD is a mechanism that adds instrumentation to the prologue and epilogue of each function that terminates in a return instruction. In the prologue, we combine the function return address with a random cookie and store the resulting RETGUARD cookie in the stack frame. In the epilogue we verify that the return address is the same one we recorded on function entry. If the addresses match, then we jump over a sequence of interrupt instructions which precede the return. If not, then the program falls through into the interrupt instructions and aborts. By inserting interrupts before the return, we mitigate against gadgets which begin shortly before the return, and larger gadgets must pass the verification process in order to jump over the interrupts and reach the return.

The random cookies used in RETGUARD are drawn from the OpenBSD *.openbsd.randomdata* section. This special read-only ELF section is pre-filled with random byte values at load time by the kernel and dynamic loader (ld.so) whenever executables or shared library objects are loaded into memory. Programs needing high quality random data can allocate memory in this section and be guaranteed that the memory will be randomized when program execution begins. RETGUARD allocates one 8 byte random cookie per function, so the RETGUARD cookie is unique per function and per call.

amd64

The RETGUARD prologue and epilogue for amd64 are shown in Figures 4 and 5. In the prologue, we fetch the function's random cookie and combine it with the return address, then store the resulting RETGUARD cookie in the stack frame. The RETGUARD cookie is calculated before frame setup, and the cookie is stored in the frame along with any other callee saved registers. Unlike the stack protector cookie, the location of the retguard cookie in the stack frame is not important, so it can be stored anywhere in the frame.

The epilogue retrieves the retguard cookie from the frame, combines it with the address we are about to return to, and compares the result with the function's random cookie. If the values match, then the jump is taken over the interrupt instructions and the function returns normally. Otherwise, the program will fall through to the interrupts and the program will abort. A representative program epilogue is shown in Figure 5

By disassembling the epilogue from each offset leading up to the return, we can verify that for each possible offset the program must either pass the random cookie check or terminate on an interrupt. Since each disassembled 'gadget' contains an interrupt instruction, gadget tooling like ROPGadget will recognize the instruction sequence as unusable, with the consequent effect that these gadgets are effectively removed from the compiled binary. In the future, should ROP tooling become clever enough to recognize the jump before

Figure 4: RETGUARD Prologue (amd64)

Instruction	Description
`mov off(%rip),%r11`	`load random cookie`
`xor (%rsp),%r11`	`xor return addr`
`push %rbp`	
`mov %rsp,%rbp`	
`push %r11`	`save retguard cookie`

Figure 5: RETGUARD Epilogue (amd64)

Instruction	Description
`pop %r11`	`load retguard cookie`
`pop %rbp`	
`xor (%rsp),%r11`	`xor return addr`
`cmp off(%rip),%r11`	`compare random cookie`
`je 2`	`jump if equal`
`int3`	`interrupt`
`int3`	`interrupt`
`retq`	

the interrupts, then the random cookie comparison will still need to be passed before the jump can be taken. In this way, RETGUARD effectively removes aligned gadgets from programs.

arm64

For arm64, the function prologue and epilogue are similar, and are shown in Figures 6 and 7. The difference between the amd64 and arm64 versions is that because arm64 is an aligned architecture, we do not need to perform the disassembly exercise for each offset leading up to the return instruction - the only instructions available as ROP gadgets are the instructions as they were intended.

Again, we see that each possible gadget in the function epilogue contains an interrupt, and will therefore be ignored by ROP gadget tooling. Should an attacker attempt to use these gadgets anyway, then the return address verification step will still need to be passed in order to bypass the interrupt. Again, the RETGUARD instrumentation effectively removes these gadgets from the binary.

Stack Protection

Finally, although the intent of RETGUARD is to make it difficult to use function return instructions as ROP gadgets, the return address verification mechanism in the epilogue has the same effect as enforcing control flow on the program. If the return address is modified on the stack, then the program will abort. This is the same effect as the existing stack canary, which is placed on the stack immediately before the return address. RETGUARD improves upon the stack canary mechanism by allocating one random cookie per function instead

Figure 6: RETGUARD Prologue (arm64)

Instruction	Description
`adrp x15, #pageoff`	`load random cookie`
`ldr x15, [x15, #off]`	`load random cookie`
`eor x15, x15, x30`	`xor return addr`
`str x15, [sp, #-16]!`	`save retguard cookie`

Figure 7: RETGUARD Epilogue (arm64)

Instruction	Description
`ldr x15, [sp], #16`	`load retguard cookie`
`adrp x9, #pageoff`	`load random cookie`
`ldr x9, [x9, #off]`	`load random cookie`
`eor x15, x15, x30`	`xor return addr`
`subs x15, x15, x9`	`compare random cookie`
`cbz x15, #8`	`jump if equal`
`brk #0x1`	`interrupt`
`ret`	

of one cookie per object file, and directly verifying the return address instead of verifying the stack canary and inferring the integrity of the return address. In this way, RETGUARD provides a stronger stack protection mechanism than simple stack canaries.

2.2 Polymorphic Gadgets

Polymorphic gadgets terminate on unintended return instructions. These gadgets do not exist on aligned architectures, but on x86, there are four bytes which decode to a return [3]: *c2*, *c3*, *ca*, and *cb*. These are shown in Table 1 along with their meanings. The four kinds of return are divided between near (c2, c3) and far (ca, cb) returns, and returns that pop additional data off the stack (c2, ca) or not (c3, cb). The most common kind of return found in ordinary programs is the c3 return, which is also the easiest kind to employ in ROP gadgets since it is a near return that does not change the current code segment or adjust the stack.

Any time any of these bytes occur in the instruction stream, they represent a potential gadget. These bytes occur in three main parts of programs:

Instruction Encoding Instructions that encode with a return

Table 1: x86 Return Instructions

Byte	Meaning
c2 *imm16*	Near return to calling procedure and pop *imm16* bytes from stack.
c3	Near return to calling procedure.
ca *imm16*	Far return to calling procedure and pop *imm16* bytes from stack.
cb	Far return to calling procedure.

Table 2: ModR/M Byte Encodings

ModR/M Byte	1st Operand	2nd Operand
c2	rax, r8	rdx, r10
c3	rax, r8	rbx, r11
ca	rcx, r9	rdx, r10
cb	rcx, r9	rbx, r11

Table 3: SIB Byte Encodings

SIB Byte	Base	Index	Scale
c2	rdx, r10	rax, r8	8
c3	rbx, r11	rax, r8	8
ca	rdx, r10	rcx, r9	8
cb	rbx, r11	rcx, r9	8

byte as part of the instruction, either as part of the instruction directly, or through the encoding of the *ModR/M* or *SIB* byte.

Constants Instructions which use a numeric constant containing a return byte, such as loading a literal value onto a register. Since OpenBSD is compiled fully PIE (position independent executable), these are always value literals, since there are no address constants.

Relocation Addresses Instructions which reference a value located in another program object such as a shared library have the locations of these objects filled in at runtime. Sometimes the location value includes a return byte.

Examples of gadgets arising from each of these program parts are shown in Figure 8, which shows for each source of polymorphic gadgets the bytes making up the intended instruction(s), what the intended instruction was, and what the gadget instructions are. In each example, the gadget bytes are highlighted for easy identification. In the first example, the unintended return instruction comes from a ModR/M byte encoding the eax / ebx register pair. In the second example, the constant value loaded into rdi contains a c3 byte. In the last example, the address of the bcmp function happens to encode with a c3 byte.

In the case of instruction encodings, the majority of unintended return instructions originate from the *ModR/M* or *SIB* byte of instructions that operate on one or two registers. For some combinations of registers the ModR/M or SIB byte will be encoded as a c2, c3, ca or cb byte, and therefore constitute a possible return. These register combinations are shown in Tables 2 and 3, where the identified registers can also be referenced by their 8 (low), 16, 32 or 64 bit aliases (*eg*. al, ax, eax, rax).

We pursue two strategies for removing polymorphic gadgets from binaries. We first attempt to transform instructions containing return bytes into equivalent instructions that do not contain any. If this is not possible, either because there is no equivalent instruction, or because it is not safe to transform, then we prepend the instruction with an alignment sled. This alignment sled is a jump instruction followed by 2-9 interrupt instructions. The intent of the alignment sled is to limit the offsets from which a gadget may start and which will terminate on the unintended return byte. By inserting several interrupt instructions before the problematic instruction, we increase the likelihood that any execution starting from an unintended offset in the instruction stream will execute an interrupt and abort.

We implemented two approaches for transforming problematic instructions into safe alternatives: Alternate register selection; and Alternate code generation. These strategies are detailed below.

Alternate Register Selection

A survey of ROP gadgets present in the OpenBSD amd64 kernel revealed that many polymorphic gadgets result from c3 bytes which encode operations on the B series of registers (rbx, ebx, bx, bl). We can therefore reduce the number of gadgets by simply reducing the use of these registers. We have modified the register allocation preference in the clang compiler to place these registers after each of the other general purpose registers, so that the B registers are assigned last. As a consequence, many functions which do not need all of the available general purpose registers will never use the B registers, and will therefore not be at risk of encoding unintended c3 bytes when operating on those registers.

This change is straightforward to implement - we simply change a list:

Before RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, **RBX**, R14, R15, R12, R13, RBP

After RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, R14, R15, R12, R13, **RBX**, RBP

This change is entirely free. There is no additional compile time cost, and no additional runtime cost. Despite being free and trivial, we shall see in Section 3 that it has a measurable effect on the number of unique gadgets present in OpenBSD binaries.

Alternate Code Generation

For instructions which do use the B series registers or other pairs of registers which can encode a return byte (as per Tables 2 and 3), or which may use a problematic constant, we have modified the clang compiler to inspect each instruction before it is emitted and attempt to exchange these problematic instruction for safe alternatives. This is the *X86FixupGadgets* pass, which identifies potential ROP gadgets and attempts to mitigate against them. An initial implementation of this pass

Figure 8: Types of polymorphic gadgets

Gadget Source	Bytes	Intended Instruction	Gadget Instruction
Instruction Encoding	`83 e3 01 01 c3`	`andl $1, %ebx` `addl %eax, %ebx`	`add %eax, (%rcx)` `ret`
Constant	`48 c7 c7 a5 c3 84 81`	`movq 0x8184c3a5, %rdi`	`movsl (%rsi), (%rdi)` `ret`
Relocation Address	`e8 95 c3 3e 00`	`callq 4113301 <bcmp>`	`xchgl %ebp, %eax` `ret`

transformed a specific subset of instructions that encoded to include c3 return bytes, and was included in OpenBSD 6.4. A more general version of this pass is being prepared for OpenBSD 6.5 that targets all four kinds of return bytes and problematic constants. This pass uses two general strategies for gadget reduction: direct instruction modification and alignment sled padding.

For instructions which include a return byte because of their particular register operand encoding, we observe that the same instruction with the register operands reversed does not result in a return byte in the emitted instruction. For example, an instruction encoding a ModR/M byte using rax as the first operand and rbx as the second operand will emit a ModR/M byte of c3, according to Table 2, but if the operands were reversed, so the first operand was rbx and the second was rax, the ModR/M byte would not encode any of the four return bytes (it would instead encode to *d8*). This relationship holds for all of the problematic register pairs identified in Tables 2 and 3 - we can always reverse the operands and emit a safe instruction. Similarly, for instructions which use only a single register operand, we can safely substitute the equivalent A series register. Our strategy for fixing instructions which encode unintended return bytes through the ModR/M or SIB bytes is therefore to insert an exchange instruction before and after the problematic instruction which swaps the values of the operand registers, and then modify the instruction to reverse the order of the operands. The effect is to perform the exact same operation as the intended instruction, but do it with the operands reversed. The resulting instructions are free of unintended return bytes and cannot terminate ROP gadgets. An example of this transformation is shown in Figure 9, which shows the original instruction bytes and intended instruction and the transformed bytes and instructions. Notice that the transformed instructions do not contain any return bytes.

For instructions which cannot be modified by exchanging their operands, or which encode constants that include a problematic byte, we insert an alignment sled before the instruction in order to interfere with gadgets which terminate on the unintended return byte. There are many reasons why we may not be able to reverse the operands used in a given instruction, such as instances when one of the registers is not a general purpose register (such as the xmm registers), the byte value is non-optional (such as the VMRESUME instruction, which encodes as *'0f 01 c3'*), the instruction is a branch or other instruction that changes control flow, or the instruction implicitly uses a register that is also one of the operands. The alignment sled is a jump instruction followed by a series of 2-9 interrupt instructions. The effect of the interrupt instructions is to cause unaligned access to the instruction stream to result in the program aborting.

Figure 10 shows the effect of inserting an alignment sled before a problematic instruction that encodes a constant with a return byte. By placing the alignment sled before the instruction, any gadgets which would have used the c3 byte as a return are constrained to avoid executing any of the interrupt instructions which precede it, with the result that unaligned execution of the instruction stream is impractical, and the c3 byte cannot be used as a return and therefore cannot be used in a ROP gadget.

3 Results

OpenBSD has applied these mitigations to the amd64 and arm64 platforms. RETGUARD has been applied to both platforms for the 6.4 release, and mitigations targeting polymorphic gadgets have been applied on the amd64 platform over the 6.3 and 6.4 releases. An enhanced version of the alternate code generation mitigation is planned for the 6.5 release.

3.1 arm64

RETGUARD was applied to the arm64 platform during the OpenBSD 6.4 release cycle. Compared to the OpenBSD 6.3 release, the number of gadgets found by ROPGadget decreased from 69935 to 46, as shown in Table 4. This decrease is attributable to the arm64 platform requiring instruction alignment, so each function protected by RETGUARD becomes effectively gadget free. The remaining 46 gadgets are all from assembly level boot code functions, which are unmapped after boot. Consequently, the OpenBSD kernel on arm64 is effectively gadget free after boot. The results in userland are much the same, with only a small number of assembly level functions contributing gadgets to userland executables and libraries. These small numbers of gadgets are generally insufficient for constructing arbitrary ROP chains,

Figure 9: Instruction transformation

Original Bytes	Original Instruction	Transform Bytes	Transform Instruction
48 89 c3	mov %rax,%rbx	48 87 d8	xchg %rbx,%rax
		48 89 d8	mov %rbx,%rax
		48 87 d8	xchg %rbx,%rax

Figure 10: Alignment sled

Original Bytes	Original Instruction	Transform Bytes	Transform Instruction
49 bc c3 f5 28 5c	mov $0x28f5c28f5c28f5c3,%r12	eb 06	jmp 6
8f c2 f5 28		cc	int3
		cc	int3
		cc	int3
		cc	int3
		cc	int3
		cc	int3
		49 bc c3 f5 28 5c	mov $0x28f5c28f5c28f5c3,%r12
		8f c2 f5 28	

Table 4: Number of Kernel Gadgets (arm64)

OpenBSD Version	# Unique Gadgets
6.3	69935
6.4	46

and so executing ROP attacks on OpenBSD binaries on the arm64 platform is generally more difficult or impossible.

3.2 amd64

ROP mitigations have been applied to the amd64 platform over several release cycles. The alternate register selection mitigation was applied for the 6.3 release. An implementation of the alternate code generation mitigation targeting some common gadget forms was applied for the 6.4 release, in addition to RETGUARD.

The alternate register selection mitigation removed approximately 6% of unique gadgets from the kernel, with negligible impact on code size and performance. The alternate code implementation removed an additional 5% of unique gadgets, at the cost of 6 bytes of additional code per transformation (which yielded an approximately 0.15% larger kernel). Since the xchg instruction is inexpensive to execute, the performance impact of this mitigation was negligible.

At the time it was applied, RETGUARD removed approximately 50% of total gadgets from the OpenBSD kernel, and around 20% of unique gadgets. The RETGUARD instrumentation adds 31 bytes per function, and increased the size of the kernel by approximately 7%. Additionally, each function reserves 8 bytes of space in the *.openbsd.randomdata* section for its random cookie, with the consequence that the random data section grows significantly compared to OpenBSD 6.3, and takes more time to fill when an executable launches. Performance overhead of RETGUARD is divided between startup cost, which is dominated by generating the random cookies for each function, and the runtime cost of executing the instrumentation in each function. For a typical system build workload, the runtime cost of RETGUARD is approximately 2%. Results summarizing the number of unique gadgets found in the OpenBSD amd64 kernel across releases is shown in Table 5. This table shows the OpenBSD version, number of unique gadgets, and kernel size for successive OpenBSD releases, with preliminary numbers shown for OpenBSD 6.5, which is currently in development. When reading this table, it is important to point out that many things were added to the kernel during each release cycle which contributed to the overall size of the kernel and the number of gadgets. For example, in the 6.4 release, RETGUARD accounted for approximately 7% of the additional code size, but the kernel grew by approximately 17%. The remaining 10% of code was new drivers and other enhancements, and this code also contributed to the overall gadget count. With this in mind, we introduce a new metric for measuring *gadget density*: unique gadgets per kilobyte. With this metric, we can estimate the effect of ROP gadget mitigations independent of the code size and more easily compare the effectiveness of gadget reduction over releases. Figure 11 shows kernel gadget density over several OpenBSD releases, including an estimate of new alternate code generation mitigations planned for OpenBSD 6.5.

In userland we found similar results, with the number of unique gadgets declining in successive OpenBSD releases. Figure 12 shows the total number of unique gadgets in the popular *sshd* binary and all linked libraries. This figure shows a re-

Figure 11: Kernel gadget density (amd64)

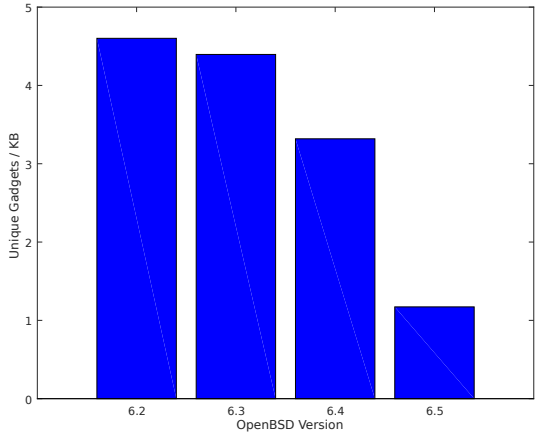

Figure 12: Number of gadgets in sshd and libraries (amd64)

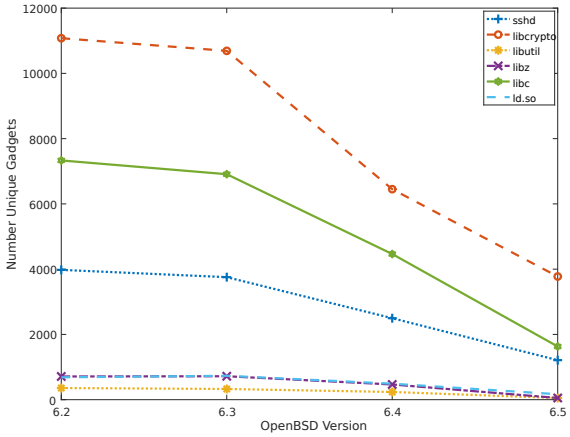

Table 5: Number of Unique Kernel Gadgets (amd64)

Version	# Unique Gadgets	Size (kB)
6.2	60589	13167
6.3	57980	13190
6.4	51229	15438
6.5	18698	15951

Figure 13: ROPGadget ropchain against OpenBSD 6.4 libc

```
$ ROPgadget.py --ropchain
               --binary OpenBSD-6.4/libc.so.92.5

Unique gadgets found: 5994
ROP chain generation
===========================================================
- Step 1 -- Write-what-where gadgets
  [-] Can't find 'mov qword ptr [r64], r64' gadget
```

duction of over 70% in the number of unique gadgets present in a typical running sshd executable between OpenBSD 6.2 and the upcoming OpenBSD 6.5.

3.3 Effect on ROP Tooling

Figure 1 showed the output from ROPGadget against the OpenBSD 6.3 libc, where we saw the tool successfully generated a ROP chain that executed a command shell using gadgets found in the libc binary. When we run the same tool against the OpenBSD 6.4 libc, we find that the tool fails to find a ROP chain that will result in the program executing a command shell. This is shown in Figure 13, where we see that after applying the mitigations described in this paper, the ROP tool is incapable of finding a write-what-where gadget, and therefore unable to construct a ROP chain that will execute a command shell. This result is true of many of the binaries and libraries included in OpenBSD 6.4, including sshd and all of it's linked libaries.

4 Conclusion

In this paper we have described a series of ROP mitigations applied in OpenBSD for both aligned and polymorphic (unaligned) ROP gadgets. For aligned gadgets, we have deployed RETGUARD on both amd64 and arm64 platforms, which resulted in a significant reduction of unique gadgets on amd64, and an almost total elimination of gadgets on arm64. For polymorphic gadgets, we have deployed a series of mitigations that eliminate gadgets either through trivial changes to register selection preferences, direct instruction modification, or forcing alignment in the instruction stream. These mitigations have resulted in a significantly reduced number of gadgets in OpenBSD binaries, both in raw numbers and in gadget density. We have shown that, as a result of these efforts, constructing ROP chains on OpenBSD is more difficult than before, and specifically shown that common ROP tools are now unable to construct ROP chains that execute a command shell against OpenBSD's libc and other binaries and libraries.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.

[3] Intel®. 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[4] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[5] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.