Let's Review Some OpenBSD Mitigations

Brian Callahan
NYC*BUG June 2026

MONMOUTH UNIVERSITY

CYBERSECURITY RESEARCH CENTER

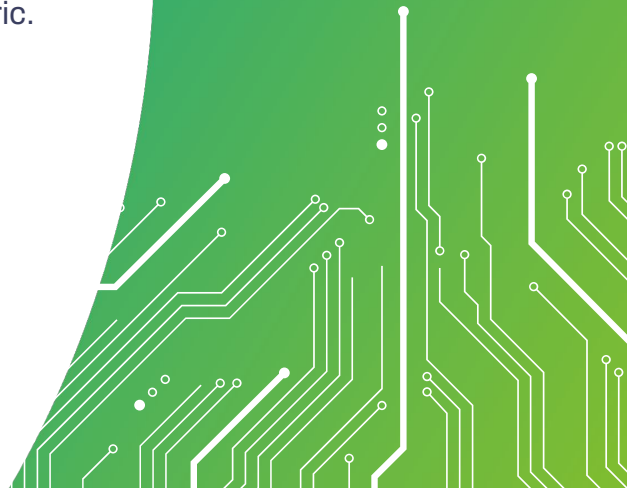
Table of contents

THE QUESTION

How do we know our security mitigations actually deliver the protections they claim they do?

Re-derive Goodhart's Law

When a metric becomes a target, it ceases to be a good metric.



01

Introduction

2019

OpenBSD developer Todd Mortimer publishes a paper in AsiaBSDCon on removing amd64 ROP gadgets in LLVM.
He developed two easy-to-implement techniques:

- Move the BX register to the end of the register allocation list: claiming **6% reduction in unique gadgets, “entirely free”**
- Rewriting potential polymorphic ROP gadget instructions: claiming **5% reduction in unique gadgets**, **0.15% binary size increase**, and **“negligible” performance impacts** (Mortimer 2019).

Let's review

Porting to FreeBSD

One way to understand the usefulness of a mitigation is to decouple it from its original context, bring it unchanged into a new context, and see if it still holds (Esposito et al. 2026).

Redoing for GCC

Another way to understand is to move the *idea*, not the *implementation*, to a new platform and see if it still holds (Callahan et al. 2026).

Three important ideas

Replication

The best ideas have a tendency to *move*, and we see this all the time with OpenBSD security innovations. So why didn't these anti-ROP techniques move?

Utility

We provide a clear, followable, testing methodology so others can validate our work. This includes a utility to perform the mitigation and a testing harness for runtime performance evaluation.

Obvious == Effective?

We want to know if this style of mitigation is worth the tradeoffs.

So how do you test this?

You get a bunch of students together to build some tools, run some experiments, and gather some data.

02

History

Developing a mental model for defenses

- Analyze current and hypothesized attack patterns
- Ask the question: ***how can we develop tools and techniques for developers that eliminate this attack pattern, or at least make it significantly more difficult to weaponize?***
- Make these tools ***cheap to use***
- Don't wait around: get it into a state where it can be tested at scale and ***put it in snapshots*** so that the larger community can act as adversarial testers (and, those who run snaps also get the benefits ASAP)

This creates a remarkably clear, consistent, and powerful mental model for defenses: reduce the intervention point (from everyone to developers) and trust your tools in their hands will increase security outward to users. This works, and OpenBSD is far from alone adopting this model.

strlcat & strlcpy

Perhaps the quintessential example:

- Real buffer overflow attacks happen because strcat and strcpy are unbounded, and if the developer doesn't meticulously check before each and every use, bad things might happen
- A tool that would go far in eliminating this class of attack is a pair of functions that do what strcat and strcpy do, and also guarantee null termination within the destination buffer's bounds: we'll call them strlcat and strlcpy
- We can audit the OpenBSD tree for all uses of strcat and strcpy, replace them with strlcat and strlcpy, and by extension ship it to all OpenBSD users for adversarial testing
    - Their computers will crash or they won't, or have some other unexpected behavior or not
- Then we can audit the ports tree and do the same

We all know how that went: used everywhere, (finally) adopted into POSIX.

It is not critique-free, but it was enough and it worked. It is also deceptively simple.

Moreover, it synced perfectly: the mental model matched the real-world attacker.

W^X

A similar story:

- When buffer overflows occur, attackers drop shellcode onto the stack and then call it, which performs arbitrary code execution (usually: launching a shell)
- Let's mark some sections of the stack writable but not executable, and other sections of the stack executable but not writable
- This kills the shellcode model directly, as it relies on stacks being both writable (to drop the shellcode onto the stack) and executable (to then call it)

W^X is now everywhere, augmented by hardware (NX bit).

Same thing: the mental model matched the real-world attacker model perfectly.

03

Bit About ROP

In the beginning, shellcode

- Find a buffer that you can overflow, preferably arbitrarily so
- Write object code (usually via hex bytes) then hijack the current function's return address on the stack, change it to point at the object code you just wrote
- Ends up creating a function that is more-or-less (in C):

```
void
shellcode_attack(void)
{
        system("/bin/sh");
}
```

- This gives the attacker a shell, now they can do (basically) anything!
- W^X basically ended this style of attack!
- So attackers needed something new...

Introducing ROP

- Find a buffer that you can overflow, preferably arbitrarily so
- Write “gadgets,” the *addresses of snippets of already existing code* from the binary itself and its shared libraries, then hijack the current function’s return address on the stack, change it to point at the first gadget
- This gives the attacker a shell, now they can do (basically) anything!

Example on next few slides (courtesy RPISEC)

Return Oriented Programming (ROP)

Now with ROP:

rax: ?

rdi: ?

rsi: ?

rdx: ?

syscall

Return Oriented Programming (ROP)

Now with ROP:

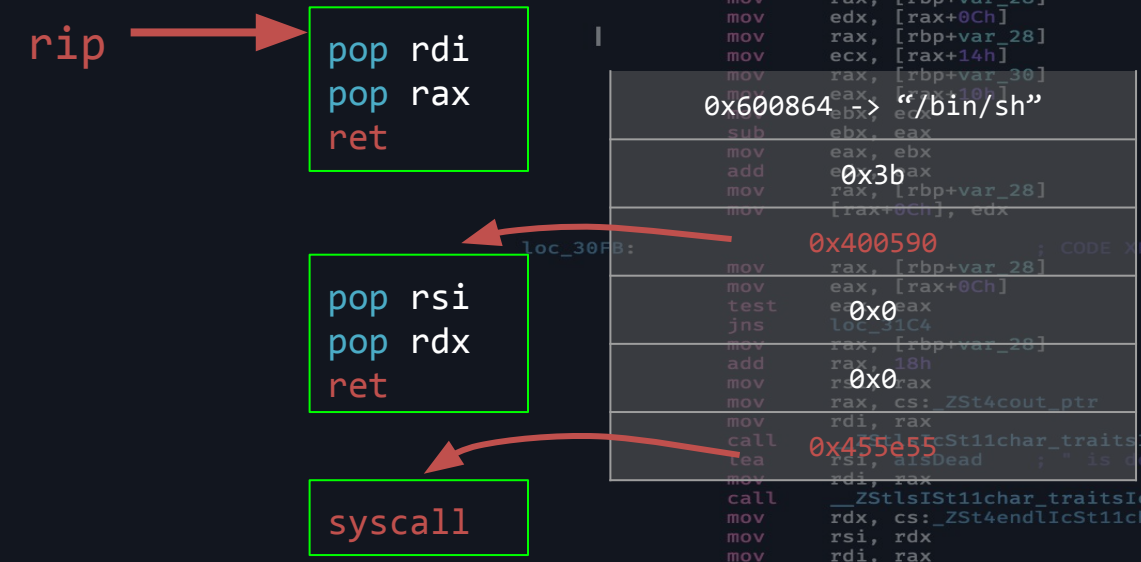
rax: ?

rdi: ?

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Now with ROP:

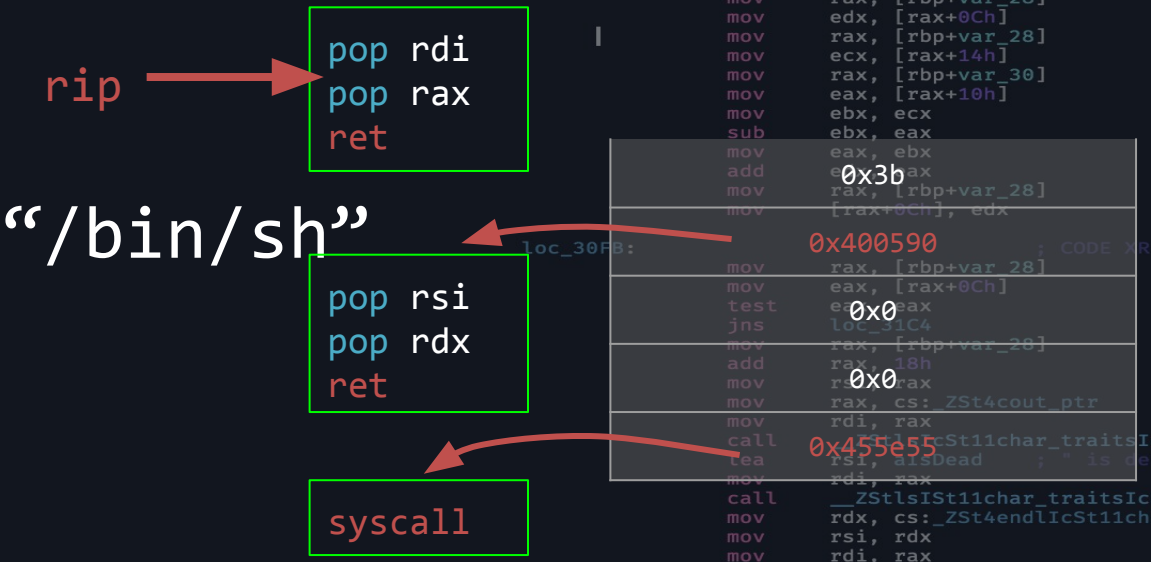
rax: ?

rdi: 0x600864 “/bin/sh”

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Now with ROP:

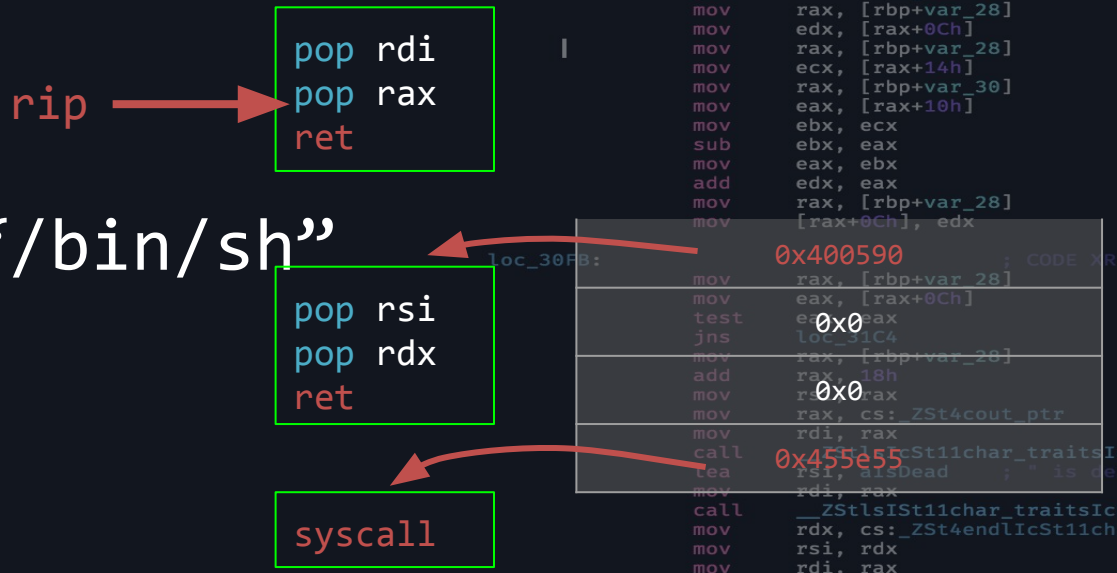
rax: 0x3b

rdi: 0x600864 “/bin/sh”

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Now with ROP:

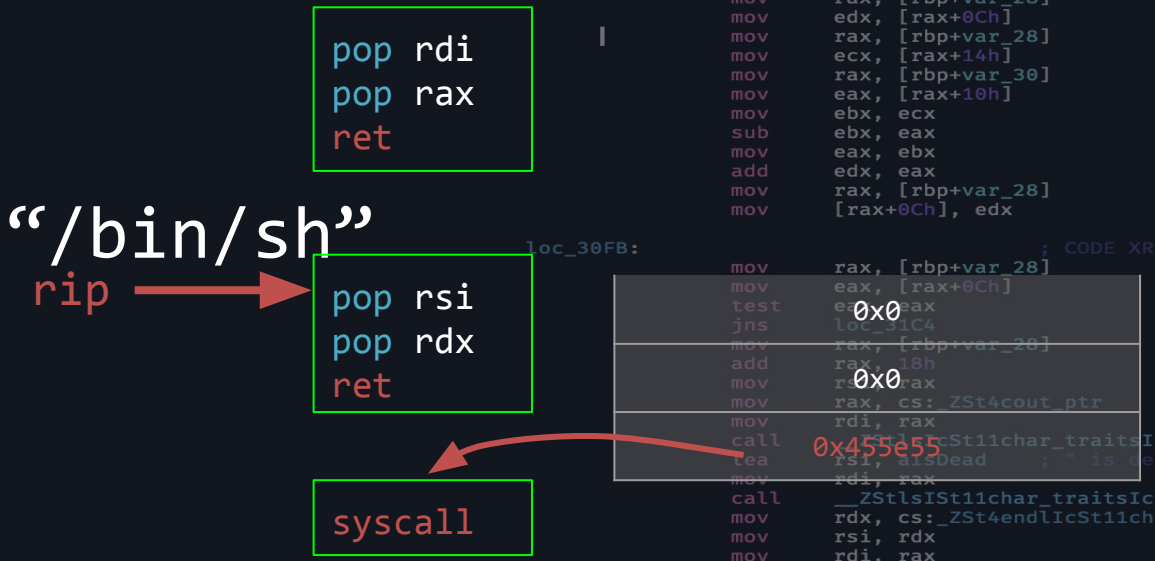
rax: 0x3b

rdi: 0x600864 “/bin/sh”

rsi: ?

rdx: ?

syscall



Return Oriented Programming (ROP)

Now with ROP:

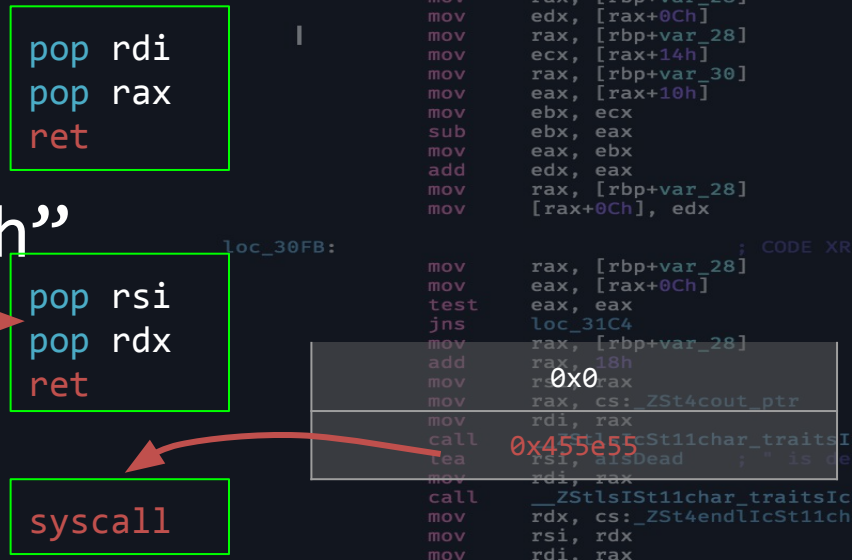
rax: 0x3b

rdi: 0x600864 “/bin/sh”

rsi: 0x0

rdx: ?

syscall



Return Oriented Programming (ROP)

Now with ROP:

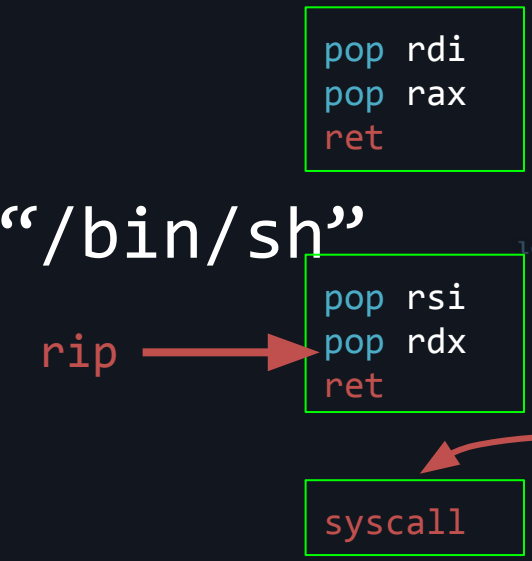
rax: 0x3b

rdi: 0x600864 “/bin/sh”

rsi: 0x0

rdx: 0x0

syscall



Return Oriented Programming (ROP)

Now with ROP:

rax: 0x3b

rdi: 0x600864 “/bin/sh”

rsi: 0x0

rdx: 0x0

syscall

Shell! :D

```
pop rdi
pop rax
ret
```

```
pop rsi
pop rdx
ret
```

rip → `syscall`

Return-Oriented Programming primer

A normal set of instructions you coded:

```
subq $256, %rcx   48 81 e9 00 01 00 00
addq %rax, %rbx               48 01 c3
```

Can become a *gadget*, if started at byte 5:

```
addl %eax, (%rax)                01 00
addb %cl, 0x1(%rax)           00 48 01
ret                              c3
```

Unaligned access and *variable-length encoding* allows for reinterpretation.

Now we have a *polymorphic gadget* that can be used in a ROP attack!

04

Let’s Review

Esposito et al. 2026

Porting and Evaluating Return-Oriented Programming Defenses Implemented by the OpenBSD Operating System — https://doi.org/10.1109/ISDFS69419.2026.11458911

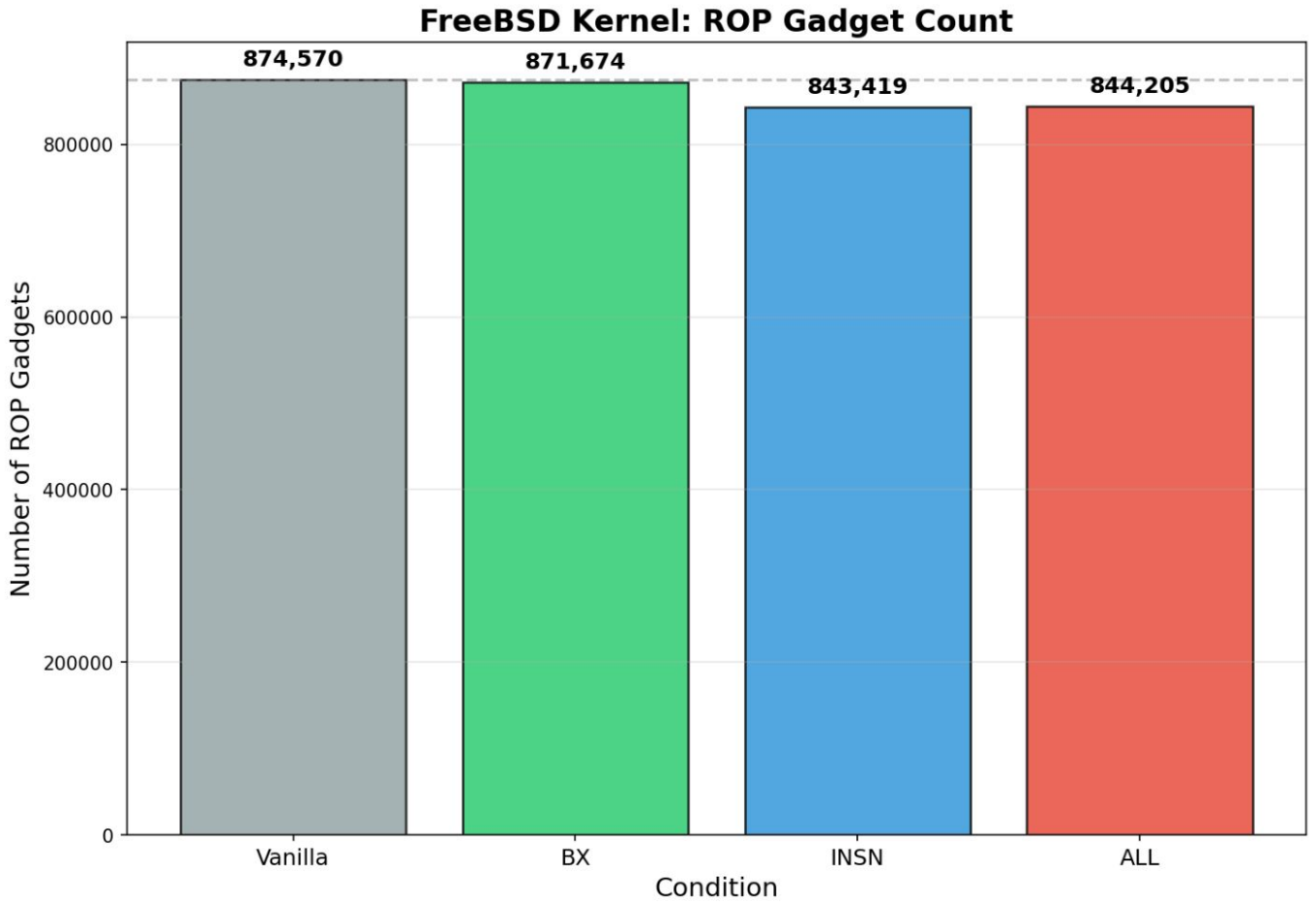
Let's directly port these mitigations from OpenBSD to FreeBSD

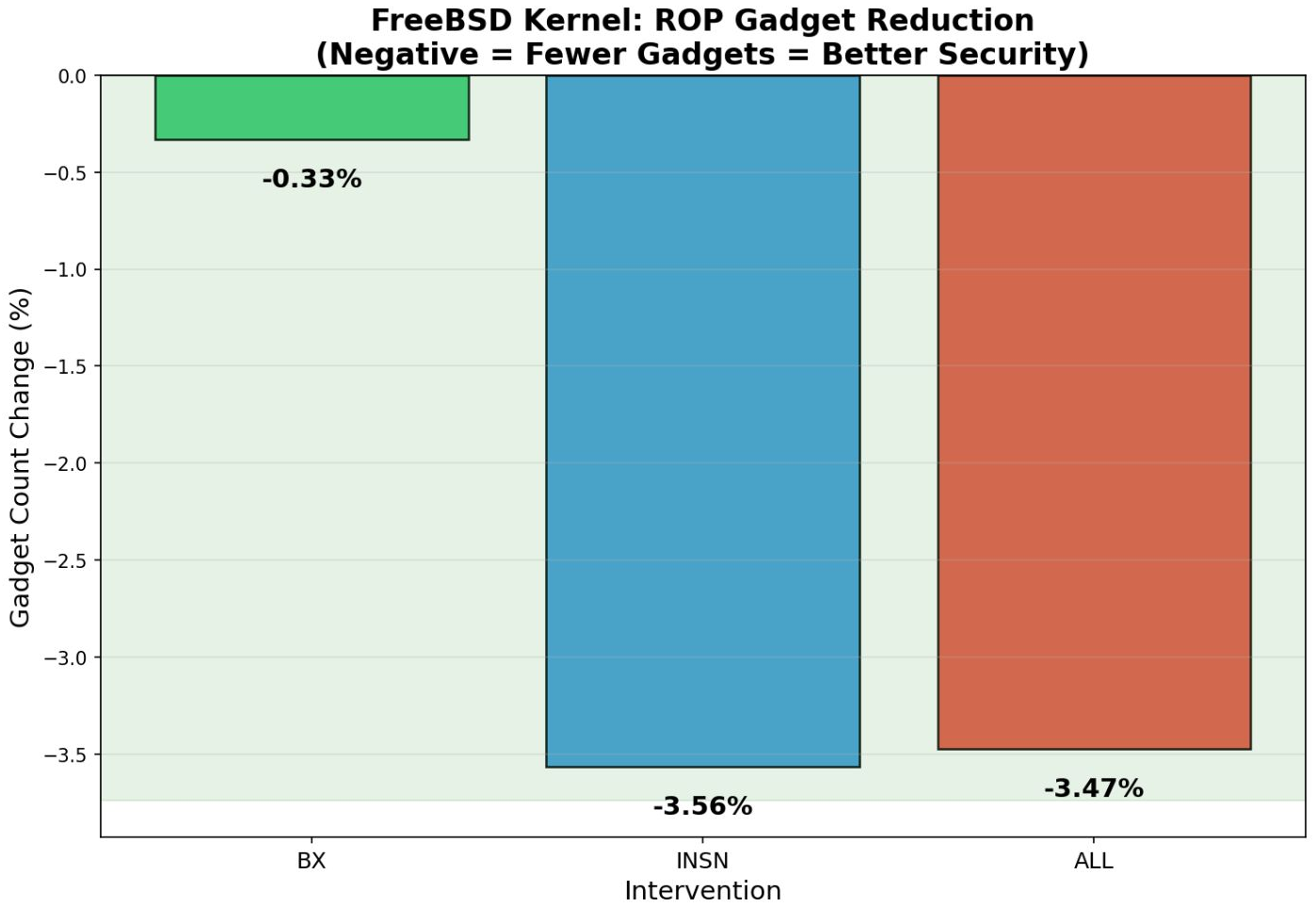
FreeBSD 14.3 and OpenBSD 7.8 use the same version of LLVM, so this is quite easy.

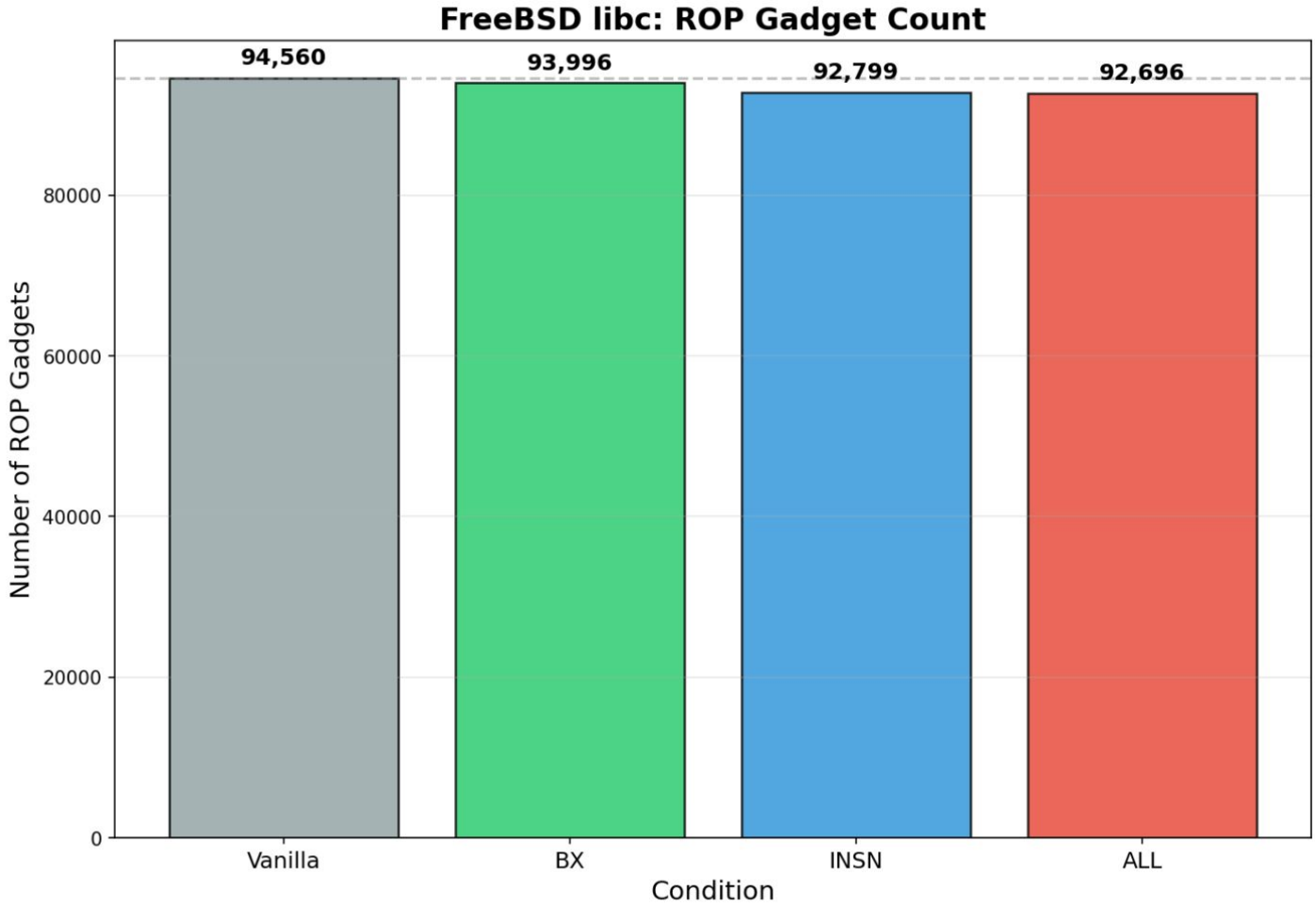
Do four test cases: unmodified FreeBSD (vanilla), only moving BX to end of register list (BX), only performing instruction rewrites (INSN), and doing both (ALL).

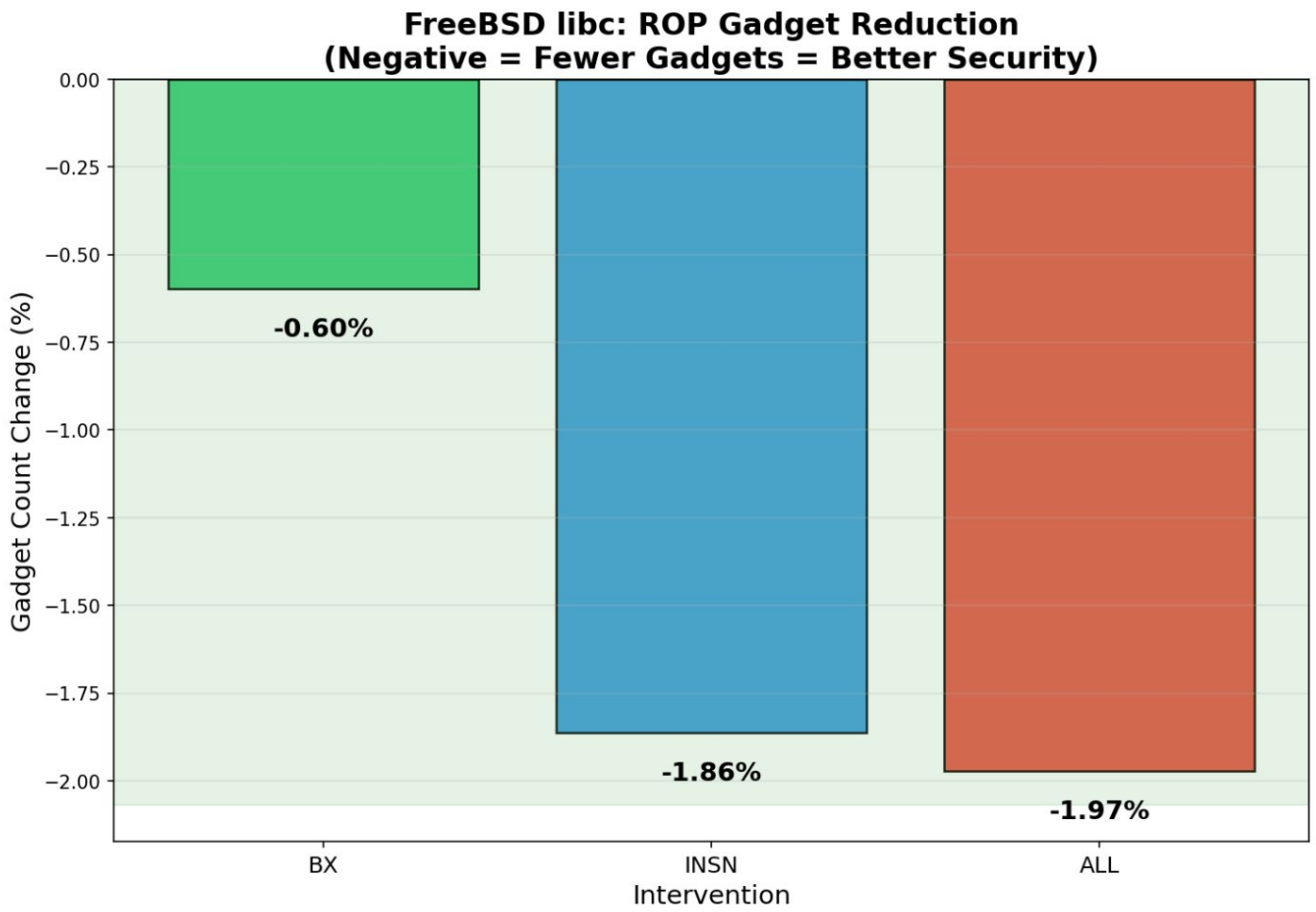
For BX, INSN, and ALL, rebuild all of FreeBSD twice, to ensure that everything gets the mitigation treatment.

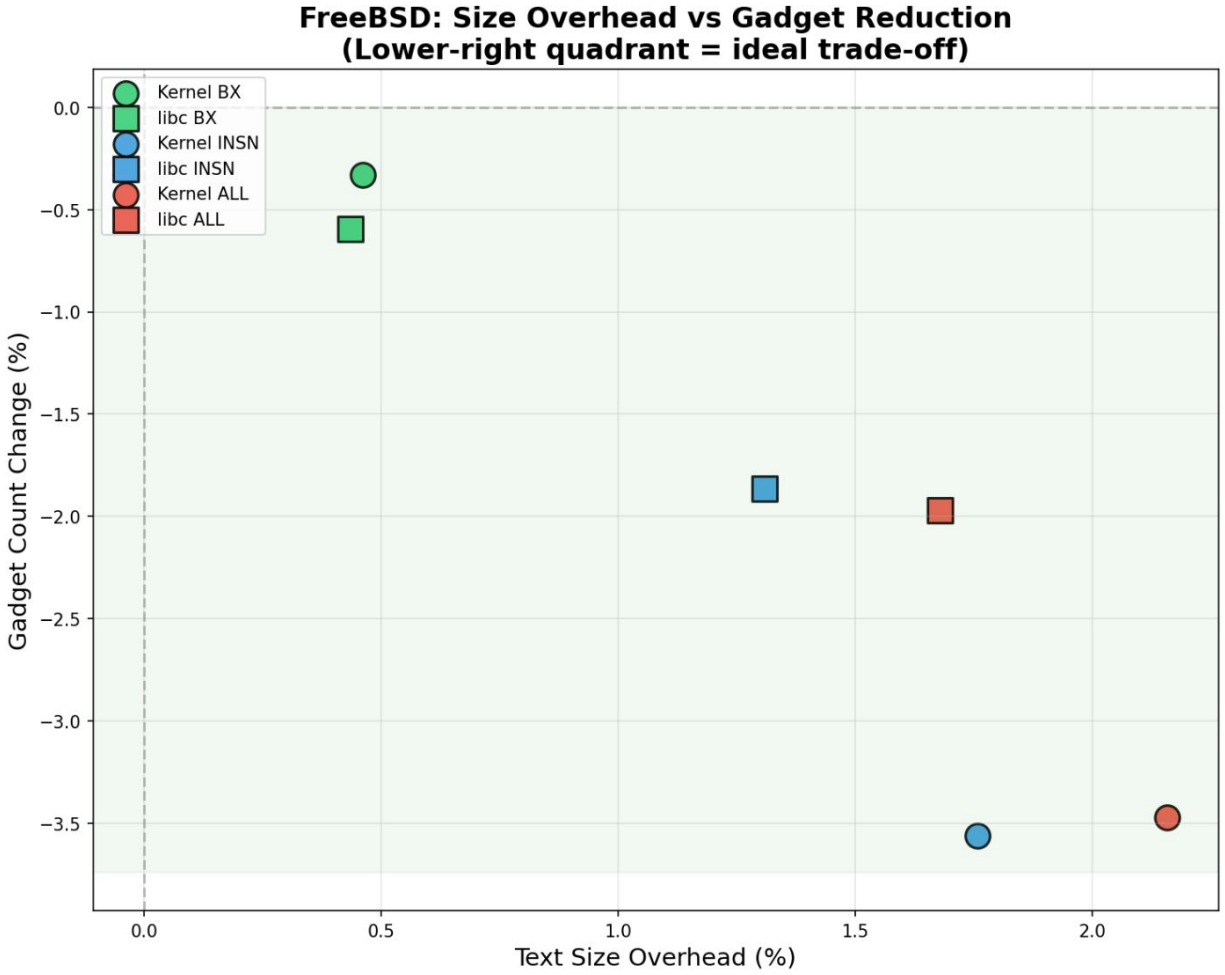
Measure everything the same way Mortimer did. While here, try to get some runtime numbers too.


FreeBSD Kernel: ROP Gadget Count
874,570
871,674
843,419
844,205
800000
600000
400000
200000
0
Number of ROP Gadgets
Vanilla
BX
INSN
ALL
Condition


FreeBSD Kernel: ROP Gadget Reduction
(Negative = Fewer Gadgets = Better Security)
Gadget Count Change (%)
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
-0.33%
-3.56%
-3.47%
BX
INSN
ALL
Intervention


FreeBSD libc: ROP Gadget Count
94,560
93,996
92,799
92,696
80000
60000
40000
20000
0
Number of ROP Gadgets
Vanilla
BX
INSN
ALL
Condition


FreeBSD libc: ROP Gadget Reduction
(Negative = Fewer Gadgets = Better Security)
Gadget Count Change (%)
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
−2.00
-0.60%
-1.86%
-1.97%
BX
INSN
ALL
Intervention


FreeBSD: Size Overhead vs Gadget Reduction
(Lower-right quadrant = ideal trade-off)
Kernel BX
libc BX
Kernel INSN
libc INSN
Kernel ALL
libc ALL
Gadget Count Change (%)
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0.0
0.5
1.0
1.5
2.0
Text Size Overhead (%)

Main takeaways

Lower gadget reduction

We got between 0.3%-3.5% gadget reduction numbers, depending on mitigation and software analyzed.

Binary size increases

Binary size *always* increased, even when just moving the BX register to the end of the register allocation list.

Runtime was a wash.

Non-obvious interplay?

With the FreeBSD kernel, ALL did *worse* than INSN alone.

A very unexpected result!

Big Takeaway

Might need to consider turning on both mitigations on a case-by-case basis. Sometimes INSN alone might do better.

Callahan et al. 2026

A Final Return for OpenBSD Anti-Return-Oriented Programming Mitigations — https://www.researchgate.net/publication/405728967_A_Final_Return_for_OpenBSD_Anti-Return-Oriented_Programming_Mitigations

Port the *idea* to GCC — tests robustness across implementations

Retry with collecting runtime data to see if it is really a wash or not

Focus exclusively on the INSN mitigation

Grab source code here: https://github.com/ibara/rop

Introducing `rop`, a peephole optimizer

Scans assembly

Line-by-line, pattern matching against known instructions that produce potential polymorphic gadgets.

Processes output assembly, not IR.

Rewrites

Matched instructions to safe equivalent

```
addq  %rax, %rbx     48 01 c3
```

becomes

```
xchgq %rax, %rbx        48 93
addq  %rbx, %rax     48 01 d8
xchgq %rax, %rbx        48 93
```

Written in D

Provides native binaries, rich and safe string handling, built into GCC (allows for complete self-hosting of experimental compiler).

Infrastructure

This paper provides a patch that makes it trivially easy to make your own standalone peephole optimizer for GCC. This is a secondary contribution.

Two compilers

Control

Using FreeBSD 15.0-RELEASE and GCC 15.2.0 with the GNU binutils, we built GCC with all support libraries (GMP, MPFR, MPC, ISL, and GNU gettext) built-in. No modifications to GCC were made.

Experimental

Using the same setup, we built a second GCC with our patch applied: doesn't change GCC itself but makes the driver call `rop` after assembly output but before the assembler.

Doubles as a test of `rop` itself—experimental GCC built without issue with `rop` in the pipeline.

A kernel and a libc, twice

Built FreeBSD kernel

Once with the control compiler and once with the experimental compiler.

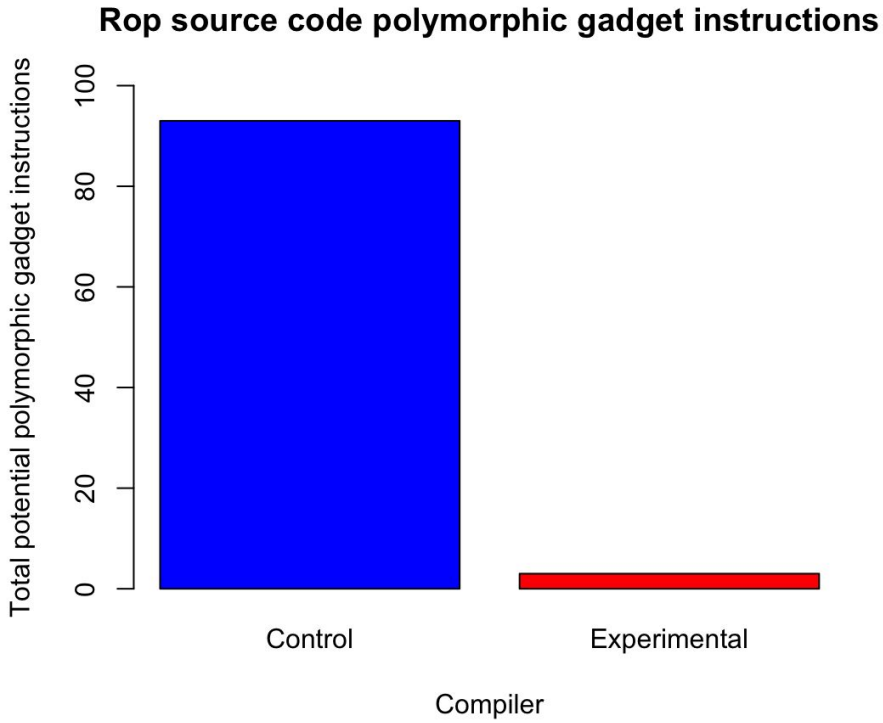
Built FreeBSD libc

Also once with the control compiler and once with the experimental compiler.

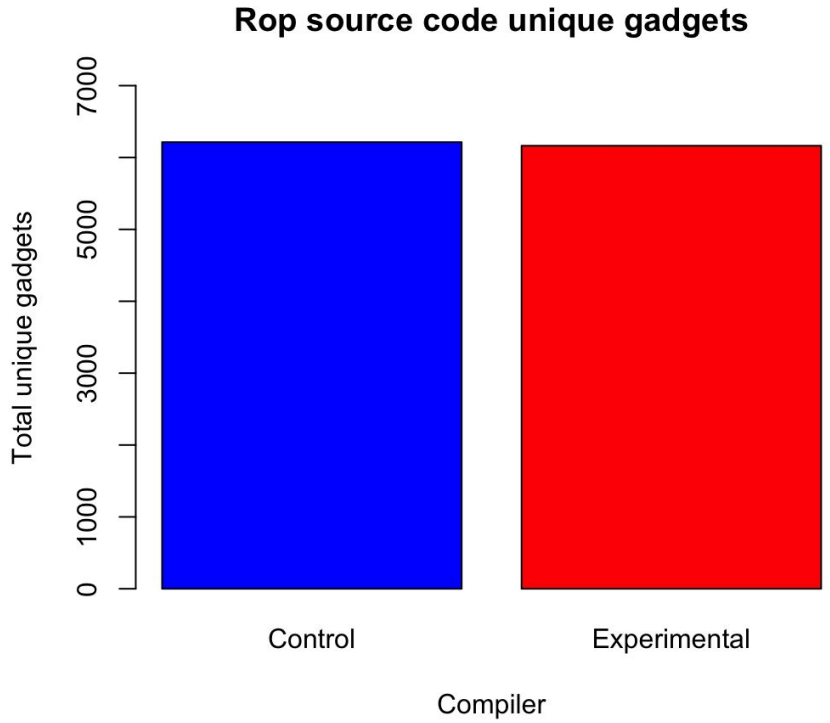
Gathered data

Unique gadget numbers with `ROPgadget` (Salwan), binary sizes with `size(1)`, and runtime statistics by running `rop` on Sqlite assembly code for billions of input lines.

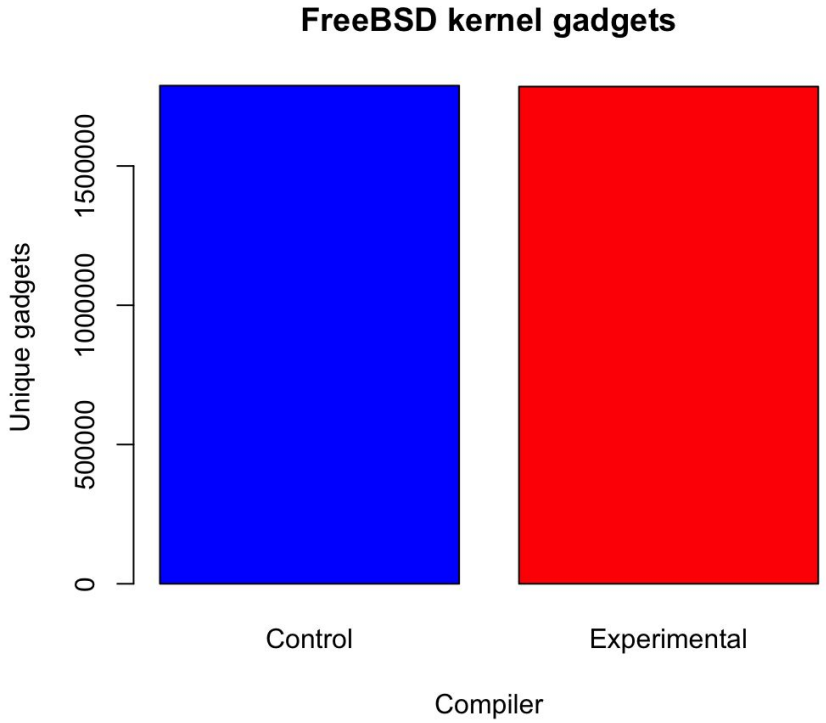
It definitely does what it sets out to do…

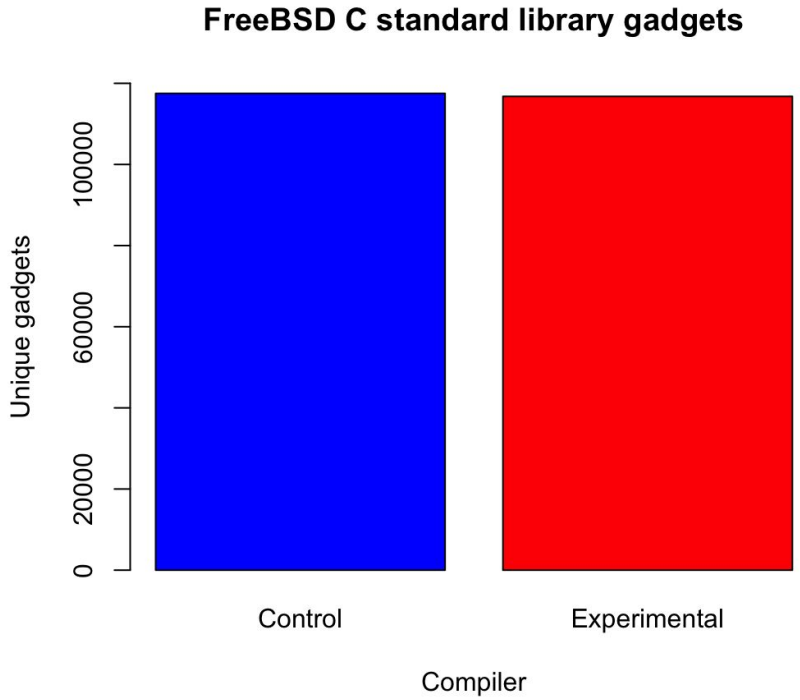


…but it still isn't great on the whole

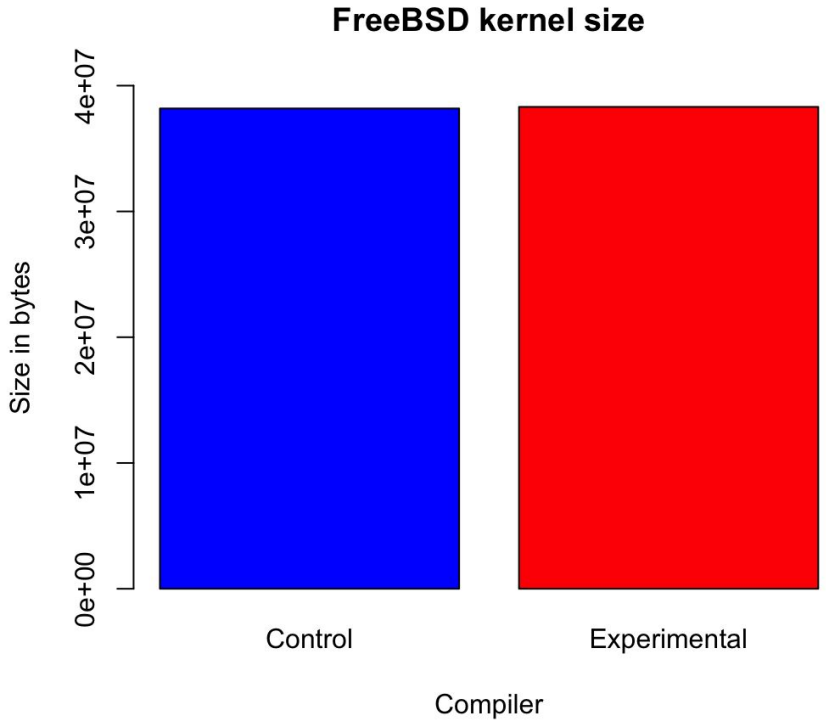


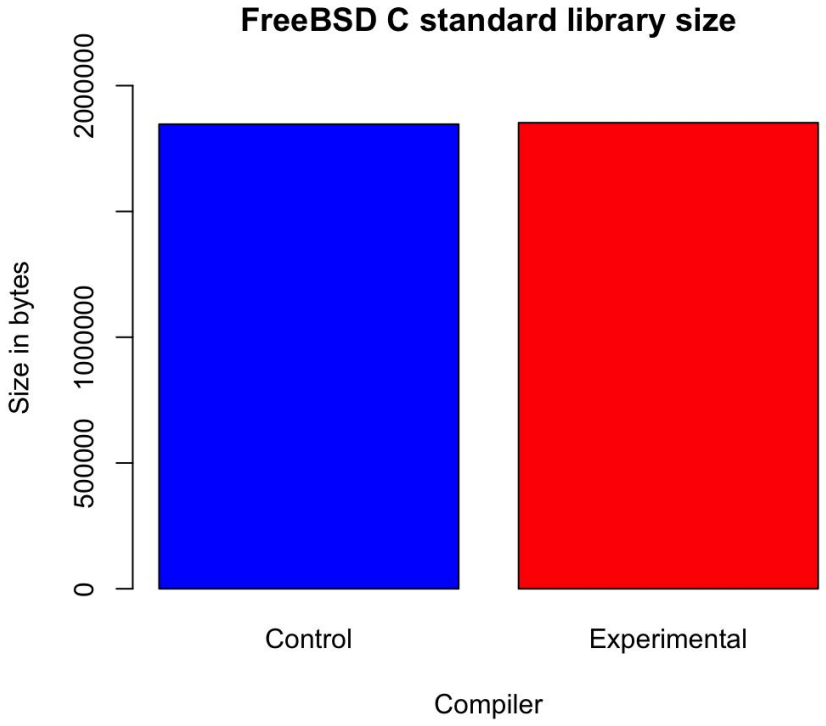
And this remains true in larger software



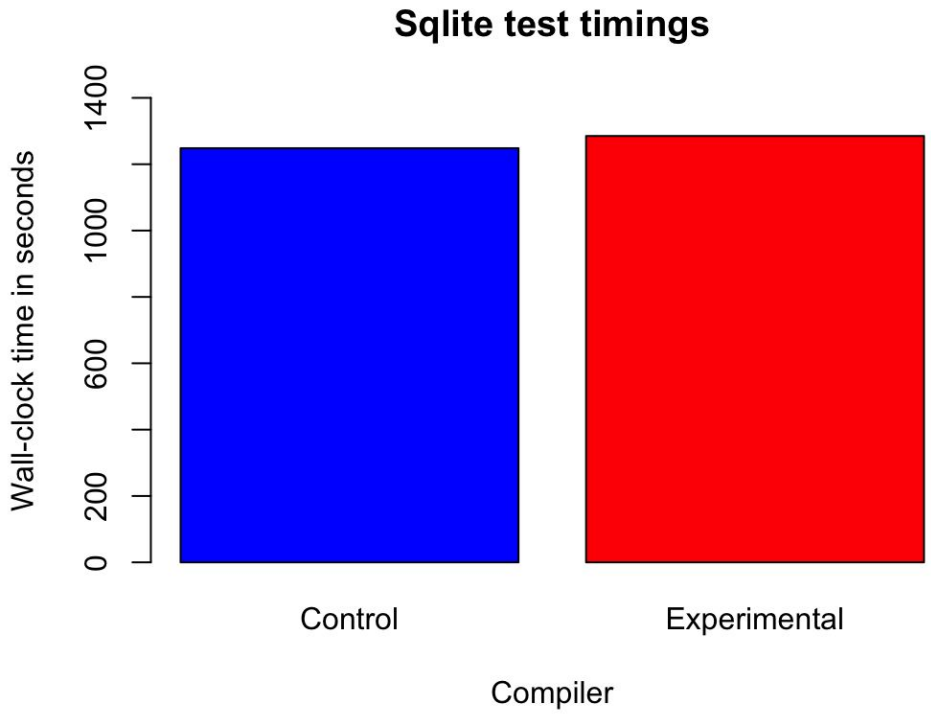


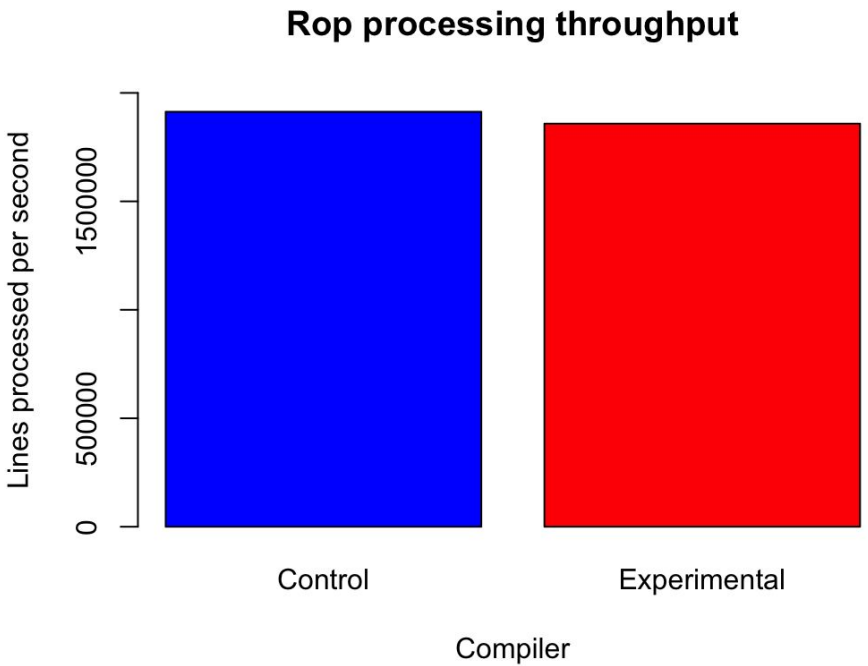
Still always grows binary size, even if only a little



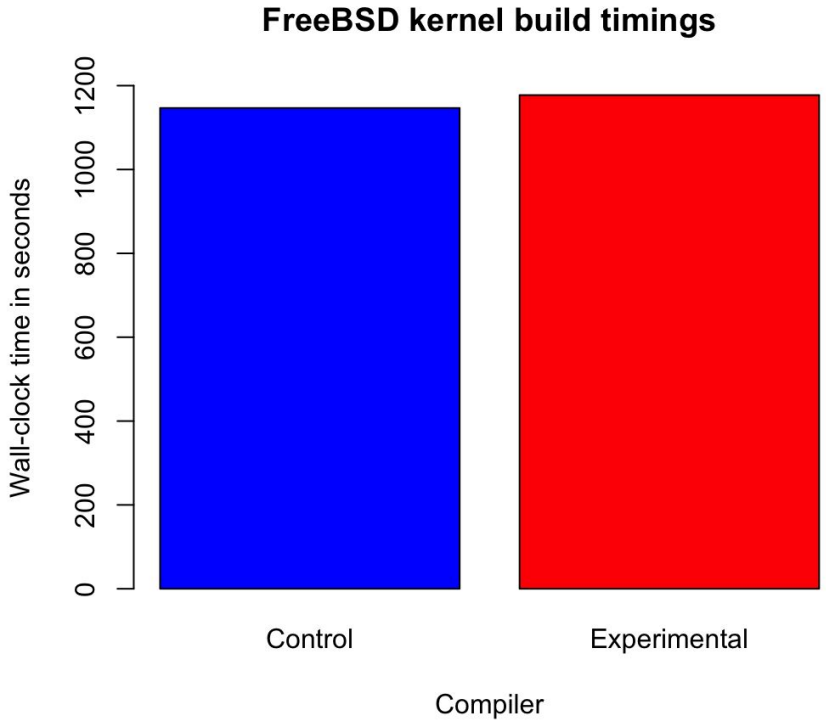


Runtime penalties are real





…and show up in real builds too



Big Takeaway

Maybe this class of mitigations is a solution in search of a problem?

05

What it means

See the trees… miss the forest

100% reduction not good enough

Even if we really did get to 0 potential polymorphic gadget instructions, it doesn't move the security needle.

`ROPgadget` not only always found a ROP chain before and after, it usually found *the same ROP chain*.

What is the mental model here?

Maybe it's not about the obvious wins. Maybe there is something else going on with the attacker model that doesn't match the mental model?

Even with 90-99% of gadgets removed, ROP still possible (Coffman et al. 2016).

06

Why it matters

It is easy (and wrong) to conclude…

- OpenBSD “got it wrong”
- Compiler-based anti-ROP techniques are entirely without merit
- We should dunk on someone because it turns out their mitigations didn’t quite hold up to external testing

None of these are the point, and you shouldn’t leave thinking any of these things!

Why it matters

- OpenBSD's mental model of attackers is largely correct
  - You don't get this long list of innovation if you're not getting it right the vast majority of the time: https://www.openbsd.org/innovations.html
- It just so happens, *for this one thing*, the mental model didn't match the attacker model
  - Turns out ROP chains are about gadget *quality*, not gadget *numbers*, and you can get a complete ROP chain with a very small number of gadgets
  - Because we focused on gadget numbers… oh look it's **Goodhart's Law**
- We now have the ability to analyze future proposed anti-ROP mitigations with a better mental model, *which helps everyone*, further demonstrating the value of the overall OpenBSD model

Improving the mental model means better security for everyone

Thanks!

Questions?
Follow me on LinkedIn! →